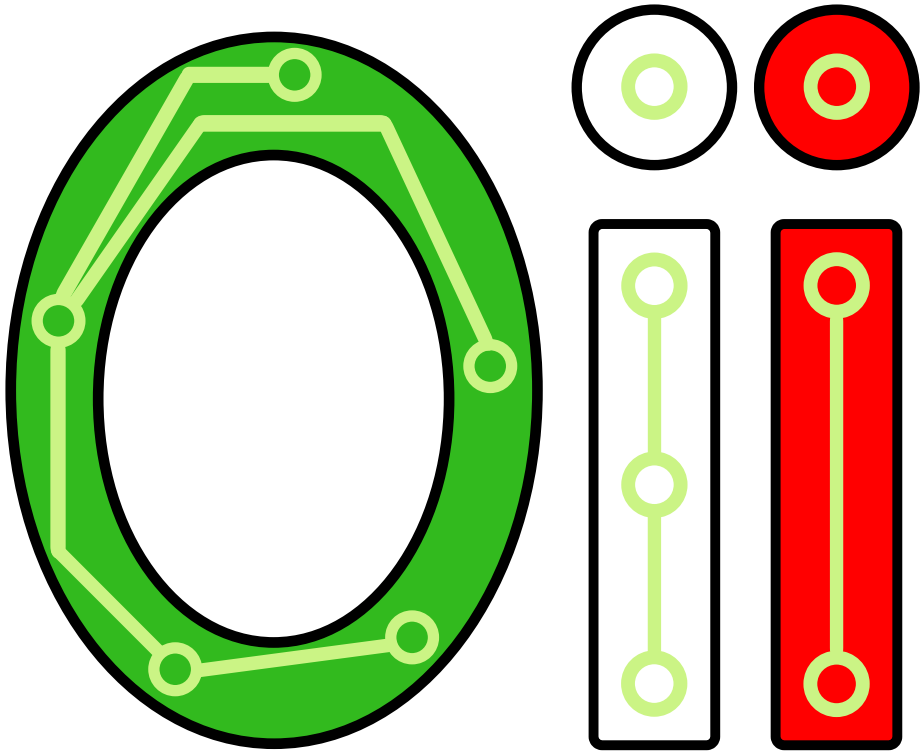
AICA
Associazione Italiana per l'Informatica
ed il Calcolo Automatico

Ministero dell'Istruzione
dell'Università e Ricerca



Olimpiadi Italiane di Informatica

Selezione Territoriale

14 Aprile 2016

Testi e soluzioni ufficiali dei problemi

Testi dei problemi

William Di Luigi, Gabriele Farina, Luigi Laura, Gemma Martini, Luca Versari

Soluzioni dei problemi

William Di Luigi, Gabriele Farina, Luca Versari

Coordinamento

Monica Gati

Supervisione a cura del Comitato per le Olimpiadi di Informatica

La spartizione di Totò (`spartizione`)

Limite di tempo:	1 secondi
Limite di memoria:	256 MiB
Difficoltà:	1

Nel film Totò Le Mokò, Totò ha un modo peculiare di dividere le gemme rubate con un suo complice:

- inizia dicendo "una a me" (e se ne prende una),
- poi dice "una a te" (e ne dà una al complice),
- poi dice "due a me" (e se ne prende **due**),
- poi dice "due a te" (ma ne dà **solo una** al complice),
- poi dice "tre a me" (e se ne prende **tre**),
- poi dice "tre a te" (ma ne dà **solo una** al complice),
- e così via...

Totò inizia *sempre* la spartizione prendendo una gemma per sé. Per esempio, se ci sono 11 gemme da spartire, Totò ne prende 8 e il suo complice 3: la prima volta ne prendono una per uno, poi Totò due e il complice una, poi Totò tre e il complice una, infine Totò prende le due rimanenti (e nessuna gemma per il complice).

La prima volta che Totò ha fatto questa spartizione il complice ha protestato, ma Totò gli ha mollato un ceffone e gli ha preso le gemme che gli aveva dato; da allora nessuno osa contraddire Totò Le Mokò in una spartizione.

Le regole della spartizione sono le stesse anche se ci sono più complici con cui dividere il bottino: ad esempio, se ci sono 16 gemme da dividere in quattro (Totò e tre complici), Totò ne prende 7 e i tre complici ne prendono 3 ciascuno: la prima volta ne prendono una per uno, poi Totò due e i complici una ciascuno, poi Totò tre e i complici una ciascuno, infine Totò prende la gemma rimanente.

Quando ci sono tante gemme Totò ha paura di sbagliarsi nella spartizione, quindi il vostro compito è quello di scrivere un programma che, ricevuti in ingresso il numero di gemme e il numero di persone (compreso Totò) tra cui spartirle, calcoli il numero di gemme che rimangono a Totò.

Dati di input

Il file `input.txt` è composto da una riga contenente G e P, due interi positivi rappresentanti rispettivamente il numero di gemme e il numero di persone (compreso Totò) tra cui spartirle.

Dati di output

Il file `output.txt` è composto da una sola riga contenente un intero positivo T: il numero di gemme che rimane a Totò dopo la spartizione.

Assunzioni

- $10 \leq G \leq 1000$.
- $2 \leq P \leq 10$.
- Nel 50% dei casi di input $P = 2$.

Esempi di input/output

input.txt	output.txt
11 2	8
18 4	9

Soluzione

Una semplice soluzione al problema consiste nell'implementare pedissequamente l'algoritmo descritto nel testo. I limiti sui valori di G e P sono tali per cui la computazione di tale algoritmo richiede pochi millisecondi sulle macchine odierne.

Esempio di codice C++11

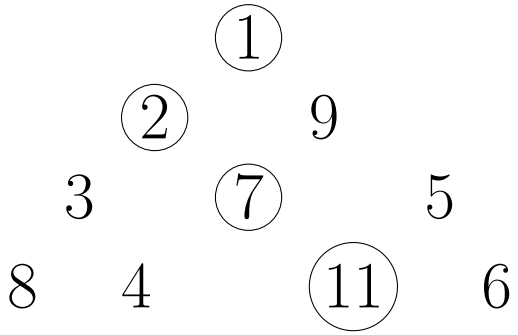
```cpp
#include <cstdio>
#include <iostream>
#include <algorithm>

int main() {
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);

  int G, P;
  std::cin >> G >> P;

  int gemme_di_toto = 0, gemme_da_spartire = G;
  int numero_turno = 1;
  while (gemme_da_spartire > 0) {
    // Totò prende tante gemme quante il numero del turno corrente, o,
    // se non ci sono abbastanza gemme, prende tutte quelle rimanenti.
    int gemme_per_toto = std::min(numero_turno, gemme_da_spartire);
    gemme_di_toto += gemme_per_toto;
    gemme_da_spartire -= gemme_per_toto;

    // Ora Totò distribuisce un massimo di P - 1 gemme ai suoi complici.
    int gemme_per_complici = std::min(P - 1, gemme_da_spartire);
    gemme_da_spartire -= gemme_per_complici;

    // Incrementa il numero del turno.
    ++numero_turno;
  }

  std::cout << gemme_di_toto << std::endl;
}
```

Discesa massima (`discesa`)

Limite di tempo:	2 secondi
Limite di memoria:	256 MiB
Difficoltà:	1

Come ben sanno gli studenti che hanno passato le selezioni scolastiche delle Olimpiadi di Informatica di quest'anno, data una piramide di numeri, definiamo una **discesa** come *una sequenza di numeri ottenuti partendo dalla cima della piramide e passando per uno dei due numeri sottostanti, fino a giungere alla base della piramide.* Inoltre, il **valore** di una discesa è definito come *la somma dei numeri della discesa.* La **discesa massima** di una piramide è quella che ha il massimo valore tra tutte le discese della piramide.

Nell'esempio seguente è stata cerchiata la discesa ottenuta partendo dalla cima scendendo prima a sinistra e poi sempre a destra fino alla base. I numeri che compongono tale discesa sono $(1, 2, 7, 11)$ e la loro somma vale 21, che è il valore di questa discesa.



La discesa massima di questa piramide è quella che si ottiene scendendo a destra, poi a sinistra e poi di nuovo a destra: i numeri di questa discesa sono $(1, 9, 7, 11)$ e la loro somma vale 28, che è il valore della discesa massima.

Il vostro compito è quello di scrivere un programma che, ricevuta in ingresso una piramide di numeri, stampi il valore della discesa massima, ovvero il massimo valore tra tutte le possibili discese della piramide.

Dati di input

Il file `input.txt` è composto da $1 + A$ righe di testo. La prima riga contiene A, un intero positivo rappresentante l'altezza della piramide. Le seguenti A righe descrivono effettivamente la piramide: l'i-esima riga (con i compreso tra 1 e A) contiene i interi positivi rappresentanti l'i-esimo "livello" della piramide.

Dati di output

Il file `output.txt` è composto da una sola riga contenente un intero positivo: il valore della discesa massima.

Assunzioni

- $1 \leq A \leq 10$.
- Il valore di ciascun numero nella piramide è un intero positivo non superiore a 100.

Esempi di input/output

Il primo esempio qui sotto si riferisce all'esempio mostrato nel testo del problema.

input.txt	output.txt
4 1 2 9 3 7 5 8 4 11 6	28
6 42 11 13 41 37 38 5 8 11 9 22 27 31 18 32 12 8 9 8 10 11	145

Soluzione

Dal momento che i vincoli sono bassi, possiamo risolvere il problema con diverse strategie. Ad esempio, possiamo enumerare tutti i possibili "modi di scendere": data l'altezza A, i modi di scendere sono tutte le sequenze binarie di lunghezza $A-1$ (possiamo infatti associare 0 = "scendi verso sinistra" e 1 = "scendi verso destra"). Per esempio, se $A = 4$, la sequenza binaria 000 significa scendere sempre a sinistra, mentre la sequenza 101 vuoldire scendere a destra, poi a sinistra e infine a destra. Questa soluzione ha un tempo di esecuzione $O(2^A)$, che è ragionevole finché A non supera (all'incirca) il valore 30.

Un'altra soluzione "inefficiente" (che però ci mette sulla buona strada per una soluzione efficiente) è utilizzare la ricorsione: definiamo una funzione f che, date le coordinate i e j della posizione corrente nella piramide, ci restituisce il valore della discesa massima possibile a partire da quella posizione. Una volta definita tale funzione ci basterà calcolare $f(0,0)$, ovvero calcolare la funzione partendo dalla cima della piramide. Tale funzione si può definire ricorsivamente nel modo seguente:

$$f(i,j) = \begin{cases} 0 & \text{se } i \geq A \text{ (ovvero se sono fuori dalla piramide)} \\ \text{piramide}_{i,j} + \max(f(i+1,j), f(i,j+1)) & \text{altrimenti} \end{cases}$$

Questa funzione è molto simile alla definizione ricorsiva della serie di Fibonacci, ed è facile vedere che il tempo di esecuzione è $O(2^A)$. Come con Fibonacci, però, possiamo aggiungere la **memoizzazione** oppure calcolare i valori partendo direttamente dalla base della piramide a salire (*bottom up*) riducendo in entrambi i casi il tempo di esecuzione a $O(A)$, che è ragionevole finché A non supera (all'incirca) il valore 10^9.

Esempio di codice C++11

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

std::vector<std::vector<int>> memo;
std::vector<std::vector<int>> piramide;
int A;

int solve(int i, int j) {
  if (i >= A) {
    // Caso base.
    return 0;
  }

  if (memo[i][j] != -1) {
    // Restituisci il vecchio risultato, se è già stato calcolato.
    return memo[i][j];
  }

  // Salva il risultato per non doverlo ricalcolare in futuro.
  int caso_migliore = std::max(solve(i + 1, j), solve(i + 1, j + 1));
  memo[i][j] = piramide[i][j] + caso_migliore;

  return memo[i][j];
}

```

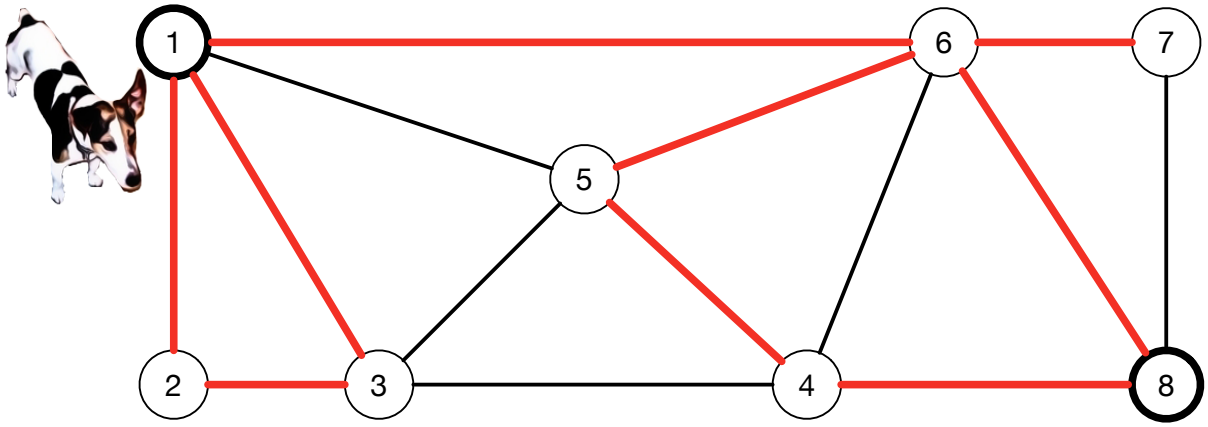
```cpp
int main() {
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);

  std::cin >> A;
  memo.resize(A);
  piramide.resize(A);

  for (int i = 0; i < A; i++) {
    for (int j = 0; j <= i; j++) {
      int x;
      std::cin >> x;
      piramide[i].push_back(x);
    }
    memo[i].resize(i + 1, -1);
  }

  std::cout << solve(0, 0) << std::endl;
}
```

Sentieri bollenti (`sentieri`)

Limite di tempo:	2 secondi
Limite di memoria:	256 MiB
Difficoltà:	1

Mojito, il piccolo cane Jack Russell mascotte delle OII, ha accompagnato Monica per la supervisione della sede di gara della finale nazionale delle Olimpiadi 2016, a Catania. Dal momento che non era troppo interessato alla disposizione dei computer, Mojito è andato a farsi una passeggiata. Adesso però il sole si è alzato e, come spesso capita in Sicilia, fa molto caldo e l'asfalto che è stato esposto al sole è bollente. Per fortuna non tutti i sentieri sono esposti al sole.

Ad esempio, nella figura sottostante, Mojito parte dal punto 1 e deve arrivare al punto 8. I sentieri bollenti sono quelli in rosso. Si può vedere che Mojito, per minimizzare il numero di sentieri bollenti può andare dal punto 1 al punto 5, da qui al 3, poi al 4 e infine al punto 8, percorrendo solo l'ultimo sentiero bollente. Altri percorsi equivalenti sono $1 \to 5 \to 3 \to 4 \to 6 \to 8$ (un solo sentiero bollente tra 6 e 8) e $1 \to 5 \to 3 \to 4 \to 6 \to 7 \to 8$ (un solo sentiero bollente tra 6 e 7).



Come si vede dall'esempio, non conta il numero complessivo di sentieri percorsi, ma solo il numero di sentieri bollenti. Il vostro compito consiste nell'aiutare Mojito a trovare una strada per tornare alla sede di gara che abbia il numero minimo di tratti esposti al sole.

Dati di input

Il file `input.txt` è composto da $1+S$ righe di testo. La prima riga contiene N, A e B, tre interi separati da spazio che rappresentano rispettivamente il numero di incroci (punti nella mappa), il numero di sentieri non bollenti, ed il numero di sentieri bollenti.

Le $A+B$ righe successive contengono due interi positivi per ogni riga, rappresentanti i punti collegati dall'i-esimo sentiero. Le prime A righe sono quelle che rappresentano i sentieri non bollenti, mentre le successive B righe rappresentano i sentieri bollenti.

Dati di output

Il file `output.txt` è composto da una sola riga contenente un intero positivo: il minimo numero di sentieri bollenti che Mojito deve percorrere per andare dal punto 1 al punto N.

Assunzioni

- Mojito parte sempre dal punto 1 e deve sempre arrivare al punto N.
- Esiste sempre almeno un percorso che collega il punto 1 al punto N.
- $5 \leq N \leq 100$.
- $10 \leq A + B \leq 1000$.
- B potrebbe valere zero.
- Un sentiero può essere percorso in entrambi i versi (informalmente: nessun sentiero è a senso unico).
- Uno stesso sentiero viene indicato al massimo una volta nel file di input.

Esempi di input/output

Il secondo esempio qui sotto si riferisce all'esempio mostrato nel testo del problema.

input.txt	output.txt
7 1 11 1 5 3 5 4 3 4 6 1 2 2 3 3 1 1 6 5 6 5 4 4 7 6 7	2
8 5 9 1 5 3 5 4 3 4 6 7 8 1 2 2 3 3 1 1 6 5 6 5 4 4 8 6 8 6 7	1

Soluzione

Semplificando, il problema ci chiede di trovare il cammino più breve sapendo che gli archi pesano 0 oppure 1. Questo problema si può risolvere efficientemente in due modi: usando l'artiglieria pesante (Algoritmo di Dijkstra) senza sfruttare il vincolo sul peso "binario" degli archi, oppure con un ragionamento che ci permette di usare una semplice BFS (leggermente modificata).

Intuitivamente, quando dobbiamo scegliere un arco da seguire, ci conviene preferire quelli di peso 0. Potremmo quindi scrivere la BFS tradizionale, ma utilizzare due code invece di una: la coda normale e la coda preferenziale (un po' come in aeroporto, per chi ha mai volato con una compagnia low cost). L'idea è quella di inserire nella coda preferenziale tutti i prossimi nodi raggiunti tramite un arco di peso 0, e nella coda normale tutti i prossimi nodi raggiunti tramite un arco di peso 1. Al momento di scegliere il prossimo nodo da visitare, dovremo fare attenzione a preferire i nodi dalla coda preferenziale (se non è vuota).

Un altro modo per vedere la stessa soluzione è usare una double ended queue (*deque*) anche detta coda doppia. Una deque, a differenza delle normali code, permette di inserire nuovi elementi sia in testa che in coda. In questo modo non è necessario controllare che la coda preferenziale sia terminata, e l'unica modifica da fare alla normale BFS è quella di inserire un controllo che decida se appendere il nuovo elemento in coda oppure in testa. Questo algoritmo è anche noto con il nome *0-1 BFS*.

Esempio di codice C++11

```cpp
#include <cstdio>
#include <iostream>
#include <vector>
#include <queue>

const int MAXN = 100;

std::vector<std::pair<int, int>> adj[MAXN];
bool visto[MAXN];

int bfs01(int N) {
  std::deque<std::pair<int, int>> coda;

  int partenza = 0;
  int arrivo = N - 1;

  coda.push_front({partenza, 0});

  while (!coda.empty()) {
    int nodo = coda.front().first;
    int peso = coda.front().second;
    coda.pop_front();

    visto[nodo] = true;

    if (nodo == arrivo) {
      return peso;
    }

```

```
30        for (auto vicino : adj[nodo]) {
31          if (!visto[vicino.first]) {
32            if (vicino.second == 0) {
33              coda.push_front({vicino.first, peso + vicino.second});
34            } else {
35              coda.push_back({vicino.first, peso + vicino.second});
36            }
37          }
38        }
39      }
40
41      // Non possiamo trovarci qui con l'esecuzione:
42      // «Esiste sempre almeno un percorso che collega il punto 1 al punto N»
43    }
44
45    int main() {
46      freopen("input.txt", "r", stdin);
47      freopen("output.txt", "w", stdout);
48
49      int N, A, B;
50      std::cin >> N >> A >> B;
51
52      for (int i = 0; i < A + B; i++) {
53        int a, b;
54        std::cin >> a >> b;
55        a--;
56        b--;
57
58        int peso = (i >= A) ? 1 : 0;
59        adj[a].push_back({b, peso});
60        adj[b].push_back({a, peso});
61      }
62
63      std::cout << bfs01(N) << std::endl;
64    }
```