OLIMPIADI ITALIANE DI INFORMATICA

UNIVERSITA' degli STUDI di ROMA TOR VERGATA

Gara di Allenamento TOR Vergata

(GATOR)

Seconda Edizione

Testi e soluzioni ufficiali

Ministero dell'Istruzione dell'Università e Ricerca

AICA
Associazione Italiana per l'Informatica ed il Calcolo Automatico

Problemi a cura di
Marco Cesati, Giuseppe F. Italiano, Luigi Laura, Massimo Regoli

Coordinamento
Monica Gati

Logo della gara
Manuela Pattarini

Testi dei problemi
Marco Cesati, William Di Luigi, Gabriele Farina, Giuseppe F. Italiano, Luigi Laura, Massimo Regoli, Luca Versari

Soluzioni dei problemi
William Di Luigi, Gabriele Farina, Luca Versari

Gestione gara online
William Di Luigi, Gabriele Farina, Luca Versari

Sistema di gara
Contest Management System (CMS)[1]

[1] http://cms-dev.github.io

Introduzione

La seconda edizione della Gara di Allenamento TOR Vergata (GATOR[2]) si è svolta sabato 11 e domenica 12 aprile 2015. La GATOR è una gara online di programmazione, con 4 problemi da svolgere in 5 ore. I problemi sono stati concepiti in modo da poter essere affrontati dagli studenti delle scuole secondarie superiori selezionati per la fase Territoriale delle Olimpiadi di Informatica. Ben 231 persone da tutta Italia si sono registrate alla seconda edizione della GATOR; di queste, 186 si sono collegate al sistema di gara e 136 hanno ottenuto punti su almeno uno dei quattro problemi proposti.

La GATOR è una iniziativa svolta nell'ambito della convenzione tra il Comitato Italiano delle Olimpiadi di Informatica e il Dipartimento di Ingegneria Civile e Ingegneria Informatica dell'Università di Roma "Tor Vergata". Le Olimpiadi di Informatica sono nate con l'intento di selezionare e formare, ogni anno, una squadra di atleti che rappresenti il nostro paese alle International Olympiad in Informatics (IOI), indette dall'UNESCO fin dal 1989. L'organizzazione delle Olimpiadi di Informatica è gestita dal Ministero dell'Istruzione, dell'Università e della Ricerca e dall'AICA, con l'obiettivo primario di stimolare l'interesse dei giovani verso la scienza dell'informazione e le tecnologie informatiche.

In questo documento trovate i testi dei quattro problemi proposti nella gara e una traccia di soluzione. I personaggi e le ambientazioni dei testi erano circoscritti al tema delle Olimpiadi, vista la candidatura di Roma alle Olimpiadi del 2024.

Ringraziamo Monica Gati, Massimo Regoli, Marco Cesati, William Di Luigi, Gabriele Farina e Luca Versari per la loro collaborazione, senza la quale non sarebbe stato possibile organizzare la GATOR. Ringraziamo inoltre Manuela Pattarini, autrice del logo.

Giuseppe F. Italiano e Luigi Laura

[2] http://www.disp.uniroma2.it/users/italiano/gator/

Classifica (`classifica`)

Limite di tempo: 1.0 secondi
Limite di memoria: 256 MiB

Difficoltà: 1

Come se le olimpiadi non bastassero, Roma ospiterà il campionato mondiale di calcio. I vertici dell'amministrazione vogliono che tutto sia perfetto, quindi hanno richiesto che venga scritto un programma che determina (data la lista delle partite giocate) la squadra vincente.

In un campionato del mondo i gironi sono composti da N squadre (N è un numero pari compreso tra 4 e 20, estremi inclusi) e queste squadre devono scontrarsi tra loro una e una sola volta. Se una squadra vince ottiene 3 punti, se pareggia 1, se perde 0.

Il regolamento stabilisce che può vincere al massimo una squadra, quindi l'amministrazione ha deciso che a parità di punteggio vincerà la squadra con numero più basso.

Dati di input

Nella prima riga del file è presente il numero N di squadre presenti nel girone. Ogni squadra è rappresentata con un numero da 1 a N. Le righe successive rappresentano tutte le partite che si sono svolte. Ciascuna riga viene presentata con 4 interi separati da spazio, rispettivamente: s_1, s_2, le due squadre che hanno giocato la partita; g_1, g_2, il numero di gol segnati da ciascuna squadra.

Dati di output

Il file `output.txt` contiene due interi separati da spazio, rispettivamente: il numero della squadra vincente, il numero di punti totalizzati dalla squadra vincente.

Assunzioni

- $4 \leq N \leq 20$.

Esempi di input/output

input.txt	output.txt
4 1 2 2 0 3 4 1 1 1 3 1 1 2 4 0 0 1 4 2 2 2 3 0 0	1 5

Note

- Per chi usa Pascal: è richiesto che si utilizzi sempre il tipo di dato `longint` al posto di `integer`.
- Un programma che stampa lo stesso output indipendentemente dal file di input non totalizza alcun punteggio.

Soluzione

Il problema richiede di trovare la squadra che occupa la prima posizione di una certa classifica. I punti in classifica per ciascuna squadra vengono calcolati in base all'esito di una serie di partite, in particolare, tutte le possibili partite tra le squadre. È un fatto noto che, date N squadre, il numero di partire che queste possono disputare (senza ripetere una partita due volte) è dato dalla formula $\frac{N(N-1)}{2}$. Una volta stilata la classifica (secondo le regole descritte nel testo), cerchiamo la squadra che ha il punteggio massimo: se ce n'è più di una, il testo dice di preferire quella con "indice" minore. Questa ricerca si può svolgere in tempo lineare (con un ciclo for oppure usando la funzione `std::max_element`) oppure facendo un apposito ordinamento (in tempo $O(N \log N)$) e prendendo poi la prima squadra.

Esempio di codice C++11

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

struct squadra_t {
    int punti;
    int indice;

    // Ritorna true se la squadra corrente è "peggiore" dell'altra
    bool operator< (const squadra_t& o) const {
        if (punti != o.punti)
            return punti < o.punti;
        else
            return indice > o.indice;
    }
};

int main() {
    // Input/output da/su file
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    int N;
    std::cin >> N;
    std::vector<squadra_t> classifica(N);

    for (int i = 0; i < N; i++) {
        classifica[i] = {0, i + 1};
    }

    for (int i = 0; i < N * (N - 1) / 2; i++) {
        int s1, s2, g1, g2;
        std::cin >> s1 >> s2 >> g1 >> g2;

        // Per comodità riportiamo i nomi delle squadre ad essere 0-based
        s1--;
        s2--;

        // Aggiorna la classifica in base alle regole
        if (g1 == g2) {
            classifica[s1].punti += 1;
            classifica[s2].punti += 1;
        } else {
            classifica[s1].punti += (g1 > g2) ? 3 : 0;
            classifica[s2].punti += (g2 > g1) ? 3 : 0;
        }
    }

    // La funzione max_element usa (dietro le quinte) l'operatore < definito sopra
    const auto& w = *std::max_element(classifica.begin(), classifica.end());
    std::cout << w.indice << " " << w.punti << std::endl;
}
```

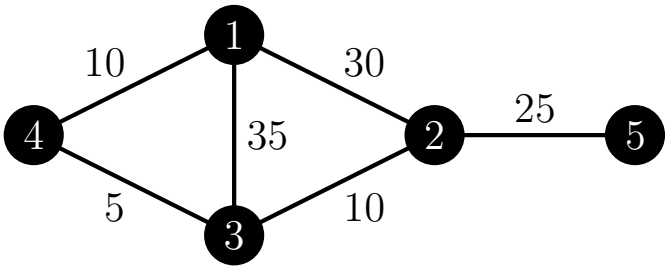
Tedoforo (`tedoforo`)

Limite di tempo:	1.0 secondi
Limite di memoria:	256 MiB

Difficoltà: 2

Luigi è un patito di fotografia, e non vuole lasciarsi scappare la possibilità di immortalare il *tedoforo* in una delle piazze di Roma. Il tedoforo è colui che porta la "teda", fiaccola cerimoniale contenente la fiamma olimpica. Come ad ogni olimpiade, il tedoforo farà una "comparsa" in ciascuna piazza della città: rimarrà in quella piazza per 10 minuti, e poi andrà (spostandosi in elicottero, per evitare il traffico romano) verso la prossima piazza.

Quindi, dal minuto 1 al minuto 10 il tedoforo si trova nella piazza 1, dal minuto 11 al minuto 20 si trova nella piazza 2, e così via. Luigi, che abita nella piazza 1, ha la possibilità di scattare lì una foto al tedoforo (perché la piazza 1 è "a 0 minuti di cammino"). Tuttavia, invece di "sprecare" la foto per la piazza 1, Luigi potrebbe decidere di spostarsi e recarsi in un'altra piazza (magari per scattare una foto migliore).



Nel caso di esempio la piazza 3 è raggiungibile in 15 minuti, quindi Luigi può raggiungerla prima che il tedoforo vada via; al contrario, per raggiungere la piazza 2 sono necessari almeno 25 minuti (è l'unica delle cinque piazze che, indipendentemente da quale strada si sceglie, non si può raggiungere prima che il tedoforo se ne sia andato!). Aiuta Luigi a capire *in quante piazze è in grado di arrivare in tempo*, in modo che possa poi decidere in quale effettivamente andare.

Dati di input

Il file `input.txt` contiene $M+1$ righe. La prima riga contiene due interi separati da spazio: N, il numero di piazze presenti a Roma; M, il numero di collegamenti tra le piazze. Le successive M righe descrivono i collegamenti. Un collegamento è definito da tre interi separati da spazio: u, v, gli indici delle due piazze collegate (gli indici vanno da 1 a N); w, il tempo necessario (espresso in minuti) a percorrere a piedi il collegamento.

Dati di output

Il file `output.txt` contiene un unico intero: il numero di piazze che Luigi è in grado di raggiungere prima che il tedoforo vada via.

Assunzioni

- $2 \leq N \leq 1000$.
- $1 \leq M \leq 10\,000$.

- Luigi "abita" nella piazza 1.
- Il tempo di percorrenza di ciascun collegamento è strettamente positivo ($w > 0$).
- I collegamenti sono bidirezionali e connettono sempre due piazze distinte ($u \neq v$).

Esempi di input/output

input.txt	output.txt
5 6 1 2 30 4 3 5 4 1 10 2 3 10 2 5 25 1 3 35	4

Note

- Per chi usa Pascal: è richiesto che si utilizzi sempre il tipo di dato `longint` al posto di `integer`.
- Un programma che stampa lo stesso output indipendentemente dal file di input non totalizza alcun punteggio.

Soluzione

Il problema chiede, dato un grafo con nodi numerati da 1 a N, di contare quanti sono i nodi u tali che la distanza minima tra 1 e u sia minore o uguale a $10u$.

■ Una soluzione con l'algoritmo di Dijkstra

Dal momento che l'algoritmo di Dijkstra[1] ci permette di calcolare in modo efficiente la distanza da un certo nodo di partenza verso *tutti* gli altri nodi del grafo, è evidente che ci basta eseguirlo (con nodo di partenza uguale a 1) e poi verificare che la distanza minima verso ciascuno degli altri nodi u sia minore o uguale a $10u$.

Esempio di codice C++11

■ Una soluzione con l'algoritmo di Dijkstra

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

const unsigned INFINITO = std::numeric_limits<unsigned>::max();
typedef unsigned vertice_t;

struct arco_t {
    vertice_t coda, testa; // I due estremi collegati
    unsigned peso;         // Il peso dell'arco
};

std::vector<std::vector<arco_t>> vicini;  // Liste di adiacenza
unsigned N, M;

struct info_t {
    vertice_t ultimo;  // Il nodo finale del cammino
    unsigned peso;     // Il peso (cumulativo) del cammino

    bool operator< (const info_t& o) const {
        return peso > o.peso;
    }
};

std::vector<unsigned> percorsi_minimi(vertice_t partenza) {
    std::vector<unsigned> distanza(N, INFINITO);
    std::priority_queue<info_t> coda;
    coda.push({partenza, 0});

    while (!coda.empty()) {
        // Cerca nella coda il cammino che conviene "continuare"
        vertice_t u = coda.top().ultimo;
        unsigned w = coda.top().peso;
        coda.pop();

        if (distanza[u] == INFINITO) {
            // Non ho mai visto il nodo u
            distanza[u] = w;

            // Visita i vicini
            for (const arco_t& arco: vicini[u]) {
                coda.push({arco.testa, w + arco.peso});
            }
        }
    }

    return std::move(distanza);
}

```

[1] https://it.wikipedia.org/wiki/Algoritmo_di_Dijkstra

```cpp
int main() {
    // Input e output da/su file
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    std::cin >> N >> M;
    vicini.resize(N);

    for (int i = 0; i < M; ++i) {
        vertice_t u, v;
        unsigned w;
        std::cin >> u >> v >> w;

        // Per comodità riportiamo i nomi dei vertici ad essere 0-based
        --u;
        --v;

        // Popola le liste di adiacenza
        vicini[u].push_back({u, v, w}); // arco u -> v di peso w
        vicini[v].push_back({v, u, w}); // arco u <- v di peso w
    }

    // Calcola le distanze minime partendo dal nodo 0
    auto distanza = percorsi_minimi(0);

    // Per ogni nodo u (da 0 a N-1), verifica che distanza[u] <= 10 * (u+1)
    unsigned risposta = 0;
    for (vertice_t u = 0; u < N; u++) {
        if (distanza[u] <= 10 * (u + 1))
            risposta++;
    }

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

Cerca le somme (`cercalesomme`)

Limite di tempo: 1.0 secondi
Limite di memoria: 256 MiB

Difficoltà: 2

Filippo, il nuovo assistente del sindaco di Roma, è molto preoccupato. Alla prima riunione in cui ha partecipato, si sono discusse le varie voci del bilancio delle olimpiadi. Lui ha trascritto tutti questi numeri su un foglio, ma poi lo ha lasciato nei pantaloni che sono finiti in lavatrice. Per fortuna non tutto è perduto: si intravedono le cifre, e lui si ricorda qual era il totale del bilancio previsto per le olimpiadi. Il vostro compito è quello di aiutare Filippo a capire quanti sono i modi di comporre le cifre, nel modo descritto di seguito, per poter ricostruire correttamente il bilancio delle olimpiadi.

Dato il foglio con le cifre, vogliamo inserire alcuni segni "`+`" in modo che il risultato delle operazioni di somma sia quello che si ricorda Filippo. Ad esempio, data la sequenza di cifre decimali:

1 2 3 4 5 6 7

Possiamo inserire quattro operatori "`+`" in modo tale che il risultato delle somme sia uguale a 100. In questo particolare caso una possibile soluzione è:

$$1 + 23 + 4 + 5 + 67$$

Aiuta Filippo a trovare tutti i possibili modi di ottenere la somma data.

Dati di input

Il file `input.txt` contiene tre righe di testo. Nella prima riga c'è un singolo numero intero positivo N che ci dice quante sono le cifre decimali nel foglio di Filippo. Nella seconda riga del file vi sono le cifre decimali separate tra loro da spazi. Nella terza riga del file c'è il totale del bilancio, ovvero il valore che deve essere ottenuto con le operazioni di somma.

Dati di output

Il file `output.txt` contiene una riga per ciascuna soluzione esatta trovata; la riga contiene le posizioni dei segni "`+`" separate da spazi. Se una posizione ha valore i, significa che il corrispondente segno "`+`" segue la i-esima cifra.

Assunzioni

- La stessa cifra può apparire più volte nella sequenza.
- Vengono date al più $N = 9$ cifre.
- È garantito che esiste almeno una soluzione.

Esempi di input/output

input.txt	output.txt
7 1 2 3 4 5 6 7 100	1 3 4 5 1 2 4 6
8 2 1 3 4 5 1 8 9 105	1 2 3 4 5 6 1 2 4 6 7
9 5 4 3 2 1 2 3 4 5 101	1 2 3 5 7 1 2 4 6 7 1 3 4 5 6 7 1 3 4 6 8 2 3 4 5 6 8 1 3 5 7 8 2 4 5 6 7 8

Note

- Per chi usa Pascal: è richiesto che si utilizzi sempre il tipo di dato `longint` al posto di `integer`.
- Un programma che stampa lo stesso output indipendentemente dal file di input non totalizza alcun punteggio.

Soluzione

Il problema chiede, date N cifre decimali ed una somma "obiettivo", in quanti modi è possibile raggruppare cifre adiacenti in modo che la somma dei numeri ottenuti sia uguale all'obiettivo. Come si può intuire dai limiti (piuttosti bassi) sul valore di N, la soluzione prevista è quella in cui si provano tutti i possibili modi di "spezzare" la sequenza di cifre.

■ Una soluzione con il backtracking

La tecnica denominata backtracking[2] consiste nel tenere traccia (tracking) di alcune informazioni sulle scelte che facciamo, per poi tentare ricorsivamente tutte le possibilità che scaturiscono da quelle scelte. Quando abbiamo finito di esplorare un ramo dell'albero delle scelte, "annulliamo" le modifiche che avevamo fatto per tenere traccia delle nostre scelte precedenti.

■ Una soluzione iterativa

Il problema si può risolvere in modo iterativo generando tutte le possibili bitmask composte da $N-1$ bit (da `000...0` a `111...1`). Ciascuna di queste 2^{N-1} bitmask, in pratica, ci dice *dove* mettere gli operatori `+` e dove non metterli.

Esempio di codice `C++11`

■ Una soluzione con il backtracking

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

const int MAXN = 9;

int sequenza[MAXN];
int N, obiettivo;
std::vector<int> soluzione;

void ricorri(int indice, int totale) {
    // Pruning (chiudi subito rami "non promettenti")
    if (totale > obiettivo) {
        return;
    }

    if (indice == N && totale == obiettivo) {
        // Stampa i valori della soluzione, separati da spazio
        std::copy(soluzione.begin(), soluzione.end() - 1,
                  std::ostream_iterator<int>(std::cout, " "));
        std::cout << std::endl;
    }

    int parziale = 0;
    for (int i = indice; i < N; i++){
        // Aggiorna la somma parziale
        parziale = parziale * 10 + sequenza[i];

        // Metti nella pila, ricorri, togli dalla pila
        soluzione.push_back(i + 1);
        ricorri(i + 1, totale + parziale);
        soluzione.pop_back();
    }
}

```

[2] https://it.wikipedia.org/wiki/Backtracking

```
37  int main() {
38      // Input e output da/su file
39      freopen("input.txt", "r", stdin);
40      freopen("output.txt", "w", stdout);
41
42      std::cin >> N;
43      for (int i = 0; i < N; i++) {
44          std::cin >> sequenza[i];
45      }
46      std::cin >> obiettivo;
47
48      // Chiama procedura ricorsiva
49      ricorri(0, 0);
50  }
```

■ Una soluzione iterativa

```
1   #include <iostream>
2   #include <vector>
3   #include <algorithm>
4   #include <iterator>
5
6   const int MAXN = 9;
7
8   int sequenza[MAXN];
9   int N, obiettivo;
10
11  int main() {
12      // Input e output da/su file
13      freopen("input.txt", "r", stdin);
14      freopen("output.txt", "w", stdout);
15
16      std::cin >> N;
17      for (int i = 0; i < N; i++) {
18          std::cin >> sequenza[i];
19      }
20      std::cin >> obiettivo;
21
22      // Prova tutti i numeri (in binario) tra 0 e 2^{N-1}-1
23      for (int mask = 0; mask < (1 << (N-1)); mask++) {
24          int totale = 0;
25          std::vector<int> soluzione;
26
27          int parziale = 0;
28          for (int i = 0; i < N; i++) {
29              parziale = parziale * 10 + sequenza[i];
30
31              // Controlla se l'i-esimo bit è acceso
32              if (mask & (1 << i)) {
33                  soluzione.push_back(i + 1);
34                  totale += parziale;
35                  parziale = 0;
36              }
37          }
38          totale += parziale;
39
40          if (totale == obiettivo) {
41              // Stampa i valori della soluzione, separati da spazio
42              std::copy(soluzione.begin(), soluzione.end(),
43                        std::ostream_iterator<int>(std::cout, " "));
44              std::cout << std::endl;
45          }
46      }
47  }
```

Canottaggio (canoa)

Limite di tempo: 1.0 secondi
Limite di memoria: 256 MiB

Difficoltà: 2

Agostino e Carmine, gli allenatori della squadra italiana di canottaggio, hanno scoperto come far rendere al meglio i loro atleti: ognuno di essi ha un valore di forza e un valore di peso e. Come è naturale, se uno massimizza la forza a bordo e minimizza il peso, la canoa procede spedita. Bisogna però tenere conto di un altro fattore: se la canoa pesa di più si immerge di più e quindi ci vuole più forza per spostarla.

Dopo varie prove, gli allenatori hanno capito che la velocità della canoa si può calcolare con una semplice formula. Indicando con f_i la forza dell'i-esimo canottiere a bordo e con p_i il suo peso:

$$v_{\text{canoa}} = \sum_i (f_i - p_i) - \frac{1}{2} \sum_i p_i$$

Aiutate Agostino e Carmine a scegliere i K atleti da far gareggiare, conoscendone i relativi valori di forza e peso.

Dati di input

Il file `input.txt` contiene $N + 1$ righe. Nella prima riga ci sono due interi separati da spazio: N, il numero di atleti totali; K, il numero di atleti che devono gareggiare. Nelle successive N righe troviamo la descrizione fisica degli atleti. Ogni atleta è descritto da due numeri interi: f, la sua forza; p, il suo peso.

Dati di output

Il file `output.txt` contiene K righe, ognuna delle quali contiene un intero: l'indice di un atleta che gareggerà.

Assegnazione del punteggio

Ciascun testcase verrà valutato con un esito che va da 0 a 1. Se la soluzione prodotta è ottimale, il punteggio per quel testcase sarà 1. Se invece gli atleti scelti producono una soluzione non ottimale, il punteggio sarà tanto più vicino a 0 quanto la soluzione è lontana dall'essere ottimale.

Assunzioni

- $1 \leq N \leq 100\,000$.
- $1 \leq K \leq N$.
- La forza ed il peso di ciascun atleta, rispettivamente, non superano il valore $1\,000\,000$.

Esempi di input/output

input.txt	output.txt
5 3 200 50 280 50 400 180 300 100 350 110	4 2 5

Note

- Per chi usa Pascal: è richiesto che si utilizzi sempre il tipo di dato `longint` al posto di `integer`.
- Un programma che stampa lo stesso output indipendentemente dal file di input non totalizza alcun punteggio.

Soluzione

Il problema chiede, dato un insieme di N atleti, di trovare un sottinsieme composto da K atleti ($K \leq N$) che massimizza una certa funzione.

Il modo più intuitivo per affrontare il problema è sicuramente quello di tentare (con un approccio "a forza bruta") tutti i possibili sottinsiemi di grandezza K, tenendo traccia di quello migliore incontrato. Questa soluzione richiede però un tempo di esecuzione proporzionale a $\binom{N}{K}$, ovvero $\frac{N!}{K!(N-K)!}$.

Studiando con attenzione la formula della velocità, notiamo che possiamo semplificarla:

$$\begin{aligned} v_{\text{canoa}} &= \sum_i (f_i - p_i) - \frac{1}{2}\sum_i p_i = \sum_i (f_i - p_i) - \sum_i \frac{1}{2} p_i \\ &= \sum_i \left(f_i - p_i - \frac{1}{2}p_i\right) = \sum_i \left(f_i - \frac{3}{2}p_i\right) \end{aligned}$$

Con questa formula si vede molto più facilmente che la velocità della canoa non è altro che la somma dei "punteggi" di ciascun atleta (dove il "punteggio" di un atleta è definito come la sua forza meno tre mezzi del suo peso). È quindi facile convincersi che, se dobbiamo scegliere un atleta, ci conviene sicuramente scegliere (in modo greedy) quello con "punteggio" maggiore.

Alla luce di questa osservazione, il nostro algoritmo sarà:

1. Ordiniamo gli atleti in base al loro "punteggio".
2. Selezioniamo (in modo greedy) i K atleti con punteggio maggiore.

Qualora ci fossero più atleti a pari merito per il K-esimo posto ci basta scegliere a caso, dal momento che il problema ammette soluzioni ottimali multiple.

Rimane solo da fare attenzione agli *errori di accuratezza dell'approssimazione* introdotti lavorando con i numeri in virgola mobile[3]. In questo caso particolare, possiamo (senza cambiare la risposta) moltiplicare per 2 il punteggio di ciascun atleta (il quale si calcolerà quindi con $2f_i - 3p_i$), lavorando così sempre e solo con numeri interi.

Esempio di codice C++11

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

int main() {
    // Input/output da/su file
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    int N, K;
    std::cin >> N >> K;

    std::vector<std::pair<int,int>> atleti;
    for (int i = 1; i <= N; i++) {
        int forza, peso;
        std::cin >> forza >> peso;
        atleti.emplace_back(2 * forza - 3 * peso, i);
    }

    std::sort(atleti.begin(), atleti.end(), std::greater<std::pair<int,int>>());
    for (int i = 0; i < K; i++) {
        std::cout << atleti[i].second << std::endl;
    }
}
```

[3] https://it.wikipedia.org/wiki/Numero_in_virgola_mobile