Selezioni Territoriali 2019 — spiegazioni e soluzioni

Di seguito spiegheremo brevemente come risolvere i quattro task delle selezioni territoriali.

Filmati e canzoni (`download`) — molto facile

Mettendo da parte i gusti strampalati di Mojito (come il fatto che stranamente tutti i filmati e tutte le canzoni in cui è interessato occupano la stessa quantità di spazio su disco) il succo del problema è questo: ci vengono dati tre numeri interi positivi, e noi vogliamo produrre due numeri interi (positivi, o eventualmente anche zero). Una volta letto e compreso il testo del problema, possiamo facilmente convincerci del fatto che la soluzione sia calcolabile partendo dal budget di byte "B" a disposizione (il primo dei 3 numeri positivi in input) e **sottraendogli ripetutamente** il valore in byte "F" dei filmati. Una volta che il budget "B" si è azzerato (o è diventato minore di F) cominciamo a fare la stessa cosa con la dimensione in byte "C" delle canzoni. Una soluzione che risolve il problema è la seguente:

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

int main() {
    int T;
    cin >> T;
    for (int caso = 1; caso <= T; caso++) {
        int B, F, C;
        cin >> B >> F >> C;

        int filmati = 0, canzoni = 0;
        for (; B >= F; filmati++) B -= F;
        for (; B >= C; canzoni++) B -= C;

        cout << "Case #" << caso << ": " << filmati << " " << canzoni << endl;
    }
}
```

Questa soluzione è più che sufficiente a ottenere tutti i punti, ma se i limiti fossero più elevati (ad esempio l'input: `1000000000000000000 7 5`) allora sarebbe necessario trovare una soluzione più efficiente. In questo task, esiste una soluzione molto efficiente (che non fa uso di cicli `for` interni, oltre a quello "obbligatorio" esterno per gestire i casi di input).

Indizio per trovarla: fai uso degli operatori di quoziente e resto 😉

Tornello olimpico (`tornello`) — facile

Prova a disegnare su un foglio di carta l'asse delle ascisse di un piano cartesiano. In parole povere è una semplice linea orizzontale in cui:

- al centro c'è il punto 0
- muovendoci dal punto 0 verso destra troviamo i punti 1, 2, 3, etc...
- muovendoci dal punto 0 verso sinistra troviamo i punti -1, -2, -3, etc...

Una volta disegnato ciò, metti la penna sul punto 0 e comincia a scansionare l'array di numeri che ti vengono dati in input: quando incontri un +1 muoviti verso destra, e analogamente muoviti verso sinistra quando incontri un -1.

Per esempio con il primo input fornito dal testo, `-1 -1 -1`, raggiungerai il punto -3 dell'asse, e la risposta per quell'input è 3. Con l'ultimo input, `-1 +1 +1 +1 -1 +1 +1 +1 -1 +1`, raggiungerai invece il punto 4... ma la risposta per quell'input è 5 🤔

Guardando l'asse disegnato sul foglio puoi osservare che per quel caso di input la **superficie totale percorsa dalla penna** è proprio uguale a 5. Con un po' di ragionamento possiamo convincerci che la risposta alla domanda del problema (ovvero: quanti studenti diversi al minimo sono stati nella stanza) è equivalente a calcolare tale superficie, o più semplicemente: la differenza tra il "massimo punto" ed il "minimo punto" raggiunti dalla penna:

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

int main() {
    int T;
    cin >> T;

    for (int t = 1; t <= T; t++) {
        int minimo = 0, massimo = 0, somma = 0, N;

        cin >> N;
        for (int i = 0; i < N; i++) {
            int x;
            cin >> x;
            somma += x;

            minimo = min(minimo, somma);
            massimo = max(massimo, somma);
        }

        cout << "Case #" << t << ": " << massimo - minimo << endl;
    }
}
```

Gerarchie di tutor (`gerarchie`) — medio

In questo problema viene richiesto di riarrangiare i tutor in modo che ognuno abbia sotto la sua supervisione solo persone con competenza inferiore a lui. Inoltre c'é il vincolo che un tutor una volta promosso non può più essere declassato.

Innanzitutto possiamo notare che esiste un ordine di arrangiamento che rispetta la seconda condizione, e consiste nell'ordinamento decrescente secondo competenza dei tutor. Questo infatti garantisce che una volta finito di promuovere un tutor non verranno mai più promossi tutor di competenza superiore alla sua, quindi lui non verrà mai declassato.

Adesso si tratta solo di fare una simulazione, guardando ogni tutor in ordine e confrontandolo con il suo diretto superiore, scambiandoli se il primo ha un valore di competenza maggiore del secondo e ripetendo il procedimento se necessario.

C'é un problema che però sorge dopo il primo scambio, ovvero che le posizioni dei tutor non sono più le stesse e perciò ricordarsi le posizioni iniziali dei tutor non basta 🤔 Un modo per ovviare a questo problema é tenere una mappa che associa un valore di competenza (che da testo sono tutti distinti) con la posizione del tutor corrispondente. Questa all'inizio avrà le associazioni competenza/tutor dell'input per poi essere aggiornata ad ogni scambio, permettendo così di conoscere in ogni momento la posizione di un tutor in base alla sua competenza. 😉

```cpp
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
using namespace std;

int gerarchie(vector<int> up, vector<int> values) {
    int N = up.size();

    vector<int> value_to_node(N);
    for (int i = 0; i < N; ++i) {
        value_to_node[values[i]] = i;
    }

    vector<int> values_order(values.begin(), values.end());
    sort(values_order.begin(), values_order.end(), greater<int>());

    int count = 0;
    for (int v : values_order) {
        int i = value_to_node[v];
        while (up[i] != -1 && values[up[i]] < values[i]) {
            int u = up[i];
            int vu = values[u];
            // scambio u con i
            value_to_node[vu] = i;
            value_to_node[v] = u;
            values[i] = vu;
            values[u] = v;
            i = u;
            count++;
        }
    }
    return count;
}

int main() {
    int T;
    cin >> T;

    for (int t = 1; t <= T; ++t) {
        int N;
        cin >> N;
        vector<int> parents(N), values(N);
        for (int i = 0; i < N; ++i) {
            cin >> parents[i] >> values[i];
        }
        cout << "Case #" << t << ": " << gerarchie(parents, values) << endl;
    }
}
```

Processori multicore (`multicore`) — difficile

Un prerequisito per risolvere questo task è conoscere la tecnica della **Programmazione Dinamica**. Se non la conosci, puoi impararla leggendo la Guida alle selezioni territoriali.

Il problema in questione è riconducibile al più classico "problema dello zaino" (*Knapsack Problem*) in particolare nella sua versione "0-1" che sta a indicare "prendo / non prendo" (nel senso che non è ammesso prendere, per esempio, solo metà di un processore dimezzandone quindi il costo ed il numero di core).

La soluzione per questo problema consiste nel modellare la risposta alla domanda (ovvero: il massimo numero di core ottenibili rientrando nel budget) con una funzione ricorsiva della forma **f(b, n)** dove:

- **b** "limita" il budget a disposizione, per esempio con b=10 stiamo provando a risolvere il problema spendendo al massimo 10;
- **n** "limita" l'insieme di CPU che vogliamo considerare, per esempio con n=5 stiamo considerando solo i primi 5 processori, nell'ordine in cui compaiono in input;

È immediato vedere che una volta che la funzione f(b, n) è pronta ed implementata, ci basta semplicemente calcolare **f(B, N)** per ottenere il risultato cercato (nota che qui le variabili in maiuscolo sono i dati di input, descritti nel testo del problema!)

Per implementare la funzione f(b, n) dobbiamo cercare, come in tutti i problemi di programmazione dinamica, una relazione di ricorrenza: un'espressione che ci permetta di calcolare f(b, n) a partire da una o più chiamate ad f() con valori dei parametri generalmente minori o uguali (naturalmente almeno uno dei parametri sarà minore, altrimenti si entrerebbe in una ricorsione infinita).

Una possibile definizione è data da:

- **f(b, 0) = 0**

  Infatti, avendo a disposizione 0 processori possiamo ottenere al massimo 0 core.

- **f(b, n) = f(b, n - 1)** *se il costo del processore n-esimo è maggiore del limite corrente b*

  Infatti, dato che il limite di budget che abbiamo imposto non è sufficiente ad acquistare l'ultima CPU della lista, tanto vale richiamare la funzione che ci dà il massimo numero di core senza però considerare l'ultima CPU!

- **f(b, n) = max(f(b - costo[n], n - 1) + numero_core[n], f(b, n - 1))** *se il costo del processore n-esimo è minore o uguale al limite corrente b*

  Infatti, dato che il limite di budget che abbiamo imposto è sufficiente ad acquistare l'ultima CPU della lista, il numero massimo di core ottenibili con il budget rimanente è dato da f(b - costo[n], n - 1) + numero_core[n]; tuttavia, non è detto che acquistare l'n-esima CPU sia la scelta giusta, per cui, "giusto per sicurezza", proviamo **anche a non prendere** tale CPU e prendiamo il massimo tra i due risultati.

Solitamente questa soluzione è più che sufficiente, ma per via dei limiti imposti nel problema (in particolare sul massimo budget possibile: `1000000000`) tale soluzione molto probabilmente andrà fuori memoria. Gli "stati" della programmazione dinamica sono infatti BxN ovvero 300000000000. Ogni stato è un numero intero a 32 bit (4 byte) quindi la memoria RAM necessaria per calcolare la soluzione al caso pessimo sarà: 300000000000 * 4 / 2^{30} = 1117.6 GB, assolutamente troppo per qualsiasi PC che si può trovare in casa (o a scuola) nel 2019 😩

Ridurre la memoria utilizzata

La soluzione descritta sopra prende dei punti, ma non tutti. Per prendere tutti i punti è infatti necessario attaccare il problema da un altro "punto di vista". Invece di spezzare la soluzione usando un *parametro* "b" che va a limitare il budget, **ribaltiamo** la formula (ﾉ°□°）ﾉ ⌒ ┻┻ e chiediamo invece qual è il minimo budget possibile che ci permette di raggiungere un determinato numero di core totali.

L'idea, quindi, è di spostare il "numero di core" da *risultato della funzione* a *parametro della funzione*, e allo stesso tempo spostare il budget nel senso opposto (da parametro a risultato). Questo trucchetto è utile solo perché il numero massimo di core totali specificato dal testo del problema (`300 * 200 = 60000`) è un numero molto più piccolo del massimo budget (`1000000000`) e quindi è **più adatto** ad essere un parametro.

Quindi, definiamo una funzione ricorsiva **g(c, n)** simile a f(b, n) ma con un parametro **c** che stavolta limita il numero di core totali. Inoltre, il valore di g(c, n) ora sta ad indicare il budget minimo richiesto per acquistare una quantità "c" di core usando i primi "n" processori. Nota bene: prima era immediato vedere che una volta implementata la funzione f(b, n) ci bastava semplicemente calcolare **f(B, N)**, ma stavolta non è immediato capire come calcolare la soluzione a partire da g(c, n). Dobbiamo infatti provare tutti i possibili valori di "c", e poi stampare quello più alto trovato tale che **g(c, N)** sia minore o uguale a **B**.

Una possibile definizione è data da:

- **g(0, n) = 0**

  Infatti, per ottenere 0 core totali ci basta anche un budget pari a 0.

- **g(c, n) = g(c, n - 1)** *se il numero di core del processore n-esimo è maggiore del limite corrente c*

  Infatti, dato che il limite sul numero di core che abbiamo imposto non ci permette di acquistare l'ultima CPU della lista, tanto vale richiamare la funzione che ci dà il minimo budget possibile per raggiungere lo stesso numero di core senza però considerare l'ultima CPU!

- **g(c, n) = min(costo[n] + g(c - numero_core[n], n - 1), g(c, n - 1))** *se il numero di core del processore n-esimo è minore o uguale al limite corrente c*

  Infatti, dato che il processore n-esimo non ci fa "sforare" il limite sul numero di core che ci siamo imposti, lo *prendiamo* e quindi usiamo un budget pari al costo di tale CPU: *sommiamo* quindi questo valore al budget necessario per ottenere i core rimanenti, che è dato da g(c - numero_core[n], n - 1); analogamente a ciò che è successo la scorsa volta, anche stavolta non è detto che acquistare l'n-esima CPU sia la scelta giusta, per cui, "giusto per sicurezza", proviamo **anche a non prendere** tale CPU e prendiamo il minimo tra i due risultati.

- **g(c, 0) = ∞** *se il limite corrente c è maggiore di 0*

  Questo è un trucco (che funziona perché stiamo usando la funzione `min`) che serve a marcare questa situazione come non valida: in parole povere stiamo dicendo alla nostra programmazione dinamica di non scegliere mai un "percorso ricorsivo" che ci porta a questa situazione (dato che *non è possibile* ottenere un numero positivo di core totali quando non ci sono più CPU disponibili!)

La soluzione in programmazione dinamica appena descritta ha un numero di stati pari a 60000x300 ovvero 18000000, per un'occupazione di RAM circa pari a: 18000000 * 4 / 2^{20} = 68.6 MB. Circa la stessa quantità di RAM utilizzata per una singola scheda del browser col quale stai leggendo questo testo 🤓

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

const int VERYBIG = 1000000001;  // just bigger than B is enough
                                 // using something too big is risky
                                 // (could overflow when adding to it...)

vector<int> costo, numero_core;
vector<vector<int>> memo;

int g(int c, int n) {
    if (c == 0) return 0;
    if (n == 0) return VERYBIG;

    if (memo[c][n] != -1) return memo[c][n];

    if (numero_core[n - 1] > c) {
        memo[c][n] = g(c, n - 1);
        return memo[c][n];
    } else {
        memo[c][n] = min(costo[n - 1] + g(c - numero_core[n - 1], n - 1), g(c, n - 1));
        return memo[c][n];
    }
}

int main() {
    int T;
    cin >> T;

    for (int t = 1; t <= T; t++) {
        costo.clear();
        numero_core.clear();
        memo.clear();

        int N, B;
        cin >> N >> B;

        costo.resize(N);
        numero_core.resize(N);

        int core_totali = 0;
        for (int i = 0; i < N; i++) {
            cin >> numero_core[i] >> costo[i];
            core_totali += numero_core[i];
        }

        memo.resize(core_totali + 1, vector<int>(N + 1, -1));  // matrice piena di -1

        for (int soluzione = core_totali; soluzione >= 0; soluzione--) {
            if (g(soluzione, N) <= B) {
                cout << "Case #" << t << ": " << soluzione << endl;
                break;
            }
        }
    }
}
```

Si può fare di meglio

La soluzione descritta sopra ovviamente è **più che sufficiente** a totalizzare tutti i punti. Tuttavia, volendo, c'è una soluzione che riduce ulteriormente la memoria (senza però velocizzare significativamente l'esecuzione) facendo uso di un solo parametro. Esercitati a trovarla!