Selezioni Territoriali 2018 — spiegazioni e soluzioni

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

Festa canina (`party`) — molto facile

Mettendo da parte i problemi di amicizia e inimicizia del famosissimo Mojito, il succo del problema è questo: ci viene dato un'array di numeri interi, e vogliamo ottenere un risultato (anch'esso un numero intero). Una volta letto e compreso il testo del problema, possiamo facilmente convincerci del fatto che la soluzione sia calcolabile come la **somma** di tutti i numeri **positivi** presenti nell'array fornito. Infatti, se un numero è negativo, vuol dire che Mojito non gradisce la presenza di quello specifico cane, quindi ci basta non invitarlo (ovvero: non includerlo nella somma). Una soluzione che risolve il problema è questa:

```cpp
int main() {
    int T;
    cin >> T;
    for (int t = 1; t <= T; t++) {
        int N; cin >> N;
        int sum = 0;
        for (int i = 0; i < N; i++) {
            int x;
            cin >> x;
            if (x > 0)
                sum += x;
        }
        cout << "Case #" << t << ": " << sum << endl;
    }
}
```

Antivirus (`antivirus`) — facile

Anche qui, una volta estratto il problema dalla "storiella", ciò che ci viene richiesto è esprimibile in termini primitivi (variabili e array). In particolare, ci vengono fornite delle variabili (numeri interi) e 4 stringhe (array di caratteri). Vogliamo calcolare 4 indici (numeri interi) che rappresentano il "punto di partenza" del virus in ciascuna stringa.

Una possibile soluzione è scrivere 5 cicli for nidificati (ovvero: uno dentro l'altro, come una matrioska). I 4 cicli più esterni faranno riferimento all'indice delle 4 stringhe (non è importante l'ordine). Per esempio, diciamo che gli indici siano: `i`, `j`, `k`, `l`. Il quinto ciclo, quello più interno, servirà invece a verificare che il virus sia effettivamente presente nelle posizioni indicate dai quattro indici. Per esempio, potremmo chiamare `m` l'indice di questo ciclo.

Il codice che segue implementa la soluzione appena descritta (con un `+ M` nella condizione dei cicli for, per evitare che i vari indici "fuoriescano" dalle stringhe).

```cpp
void solve(int t) {
    int N1, N2, N3, N4;
    cin >> N1 >> N2 >> N3 >> N4;

    int M;
    cin >> M;

    string F1, F2, F3, F4;
    cin >> F1 >> F2 >> F3 >> F4;

    for (int i = 0; i + M <= N1; i++)
    for (int j = 0; j + M <= N2; j++)
    for (int k = 0; k + M <= N3; k++)
    for (int l = 0; l + M <= N4; l++) {
        bool match = true;

        for (int m = 0; m < M; m++)
            if (F1[i + m] != F2[j + m] || F2[j + m] != F3[k + m] || F3[k + m] != F4[l + m])
                match = false;

        if (match) {
            cout << "Case #" << t << ": " << i << " " << j << " " << k << " " << l << endl;
            return;
        }
    }
```

```
}

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

    for (int t = 1; t <= T; t++) {
        solve(t);
    }
}
```

ATTENZIONE! Questa soluzione, sebbene sia sufficiente a prendere tutti i punti, non è quella asintoticamente ottimale. Infatti, la complessità computazionale di questa soluzione, quando la esprimiamo come funzione della lunghezza massima `N` delle quattro stringhe e della lunghezza massima `M` del virus, diventa pari a: $O(N^4M)$.

Dato che i valori massimi di `N` e `M` sono piuttosto bassi, il nostro programma terminerà la sua esecuzione in un tempo ragionevole. Tuttavia, se questo stesso problema fosse stato dato alla *fase nazionale*, i valori sarebbero sicuramente stati molto più elevati. Esistono infatti soluzioni asintoticamente molto più efficienti di quella appena descritta!

Radioanalisi fossile (`xray`) — medio

Un modo per risolvere questo problema è guardando ai "picchi". Possiamo convincerci che un algoritmo ottimale (in termini di: numero di azionamenti della macchina) è il seguente:

1. cerchiamo il massimo nell'array, ad esempio otterremo `8` se l'array fosse `5 8 6`;
2. lo riduciamo fino a che non diventa uguale al più grande dei suoi vicini, nel nostro caso `8` si riduce a `6` mediante 2 azionamenti;
3. torniamo al punto 1

Naturalmente dobbiamo fare attenzione al caso in cui, per esempio, l'array fosse `5 8 8 6`. In questo caso infatti il massimo non è un solo elemento ma un intervallo di due elementi: dobbiamo quindi considerarli tutti come fossero un unico elemento (ed applicare le radiazioni all'intero intervallo). Dobbiamo inoltre fare attenzione a terminare l'algoritmo non appena tutti gli elementi diventano pari a zero.

Un'implementazione alternativa di questa soluzione (più facile da codificare) considera tutti i numeri, da sinistra a destra, riducendoli a zero sequenzialmente (uno dopo l'altro) cercando di "includere" quanti più vicini possibile (ovvero: i vicini a destra fino a che non si incontra uno zero o la fine dell'array). È ragionevole convincersi che questa soluzione è equivalente a quella descritta sopra.

```
const int MAXN = 1000;

int compute(int N, int vec[MAXN + 2]) {
    int risp = 0;
    for (int i = 1; i <= N; i++) {
        while (vec[i]) {
            risp++;
            for (int j = i; j <= N; j++) {
                if (!vec[j]) break;
                vec[j]--;
            }
        }
    }

    return risp;
}

void solve(int t) {
    int N;
    cin >> N;
    int vec[MAXN + 2];

    vec[0] = vec[N + 1] = 0;
    for (int i = 1; i <= N; i++) {
        cin >> vec[i];
    }

    cout << "Case #" << t << ": " << compute(N, vec) << endl;
}

int main() {
    int T;
```

```
    cin >> T;

    for (int t = 1; t <= T; t++) {
        solve(t);
    }
}
```

Escursione (`escursione`) — difficile

Per risolvere questo problema è necessario avere dimestichezza con la ricorsione (per risolvere il problema in modo subottimale) o con i grafi (per la soluzione ottimale).

Esplorazione completa

Una possibile soluzione subottimale è utilizzare una funzione ricorsiva che "esplora" la matrice. La funzione, dato un punto di partenza, visita ricorsivamente tutte le direzioni possibili (sinistra, sopra, destra, sotto) a patto che non escano fuori dai bordi. La funzione dovrà quindi avere tra i suoi parametri: la posizione `i` e `j`, la differenza di altitudine più alta trovata fin ora.

Affinché la funzione non vada in ciclo infinito, è necessario mantenere globalmente una matrice booleana che indica se abbiamo visitato oppure no ciascuna cella. Dovremo quindi fare attenzione a impostare a `true` la posizione in cui stiamo entrando ricorsivamente, e soprattutto di reimpostare a `false` la posizione dalla quale stiamo uscendo ricorsivamente. Questa tecnica prende il nome di backtracking.

Nel corpo della funzione dobbiamo controllare se la posizione attuale corrisponde con quella di arrivo e, se è così, dobbiamo memorizzare in una variabile globale il risultato migliore trovato fin ora (utilizzando il parametro della funzione che indica la differenza di altitudine più alta del percorso).

Questa soluzione è corretta perché stiamo percorrendo tutti i possibili percorsi che partono dalla posizione di partenza ed arrivano a quella di arrivo senza passare due volte per lo stesso punto. Il problema è: **quanto tempo impiega?** La risposta è: **TROPPO!** In una griglia 100x100 non basterebbero tutte le ore di gara a calcolare la soluzione. In realtà, non basterebbe nemmeno un milione di anni! Un'illustrazione di questo fenomeno (che prende il nome di *crescita esponenziale*) si può trovare in questo simpatico video.

Ottimizziamo!

Un modo per ottimizzare questa soluzione è memorizzare per ciascuna posizione (oltre allo stato "visitato" / "non visitato") anche un'altra informazione: la migliore soluzione trovata per arrivare dal punto di partenza a quella posizione. Infatti, mantenendo questa informazione, possiamo **terminare anticipatamente** un intero ramo ricorsivo se notiamo che la soluzione "candidata" (quella che ci stiamo portando dietro nel parametro della funzione ricorsiva) è maggiore di quella che abbiamo salvato globalmente nella posizione corrente. Un accorgimento a cui dobbiamo far caso è che all'inizio del programma dobbiamo inizializzare la miglior soluzione trovata per ciascuna posizione ad un valore molto alto (è prassi definire una costante `INF` nel codice e impostarla a un valore sicuramente più alto della soluzione cercata).

Questa soluzione dovrebbe essere molto più veloce. Probabilmente è sufficiente a prendere tutti i punti.

Ottimizziamo ancora!

La soluzione ottimale a questo problema consiste nell'usare una variazione dell'Algoritmo di Dijkstra.

```
#define INFTY 1000000000

const int MAXH = 100;
const int MAXW = 100;

int A[MAXH + 2][MAXW + 2];
int D[MAXH + 2][MAXW + 2];
int H, W;

int compute() {
  priority_queue<pair<int, pair<int, int>>> q;

  q.push(make_pair(0, make_pair(1, 1)));

  while (!q.empty()) {
    auto top = q.top();
    q.pop();

    if (D[top.second.first][top.second.second] != INFTY)
      continue;
```

```
    D[top.second.first][top.second.second] = -top.first;

    for (int i = -1; i <= 1; i++) {
      for (int j = -1; j <= 1; j++) {
        if (i * j == 0) {
          q.push(make_pair(
              -max(-top.first,
                   abs(A[top.second.first][top.second.second] -
                       A[top.second.first + i][top.second.second + j])),
              make_pair(top.second.first + i, top.second.second + j)));
        }
      }
    }
  }
  return D[H][W];
}

void solve(int t) {
  cin >> H >> W;

  memset(D, 0, sizeof(D));


  for (int i = 1; i <= H; i++) {
    for (int j = 1; j <= W; j++) {
      cin >> A[i][j];
      D[i][j] = INFTY;
    }
  }

  cout << "Case #" << t << ": " << compute() << endl;
}

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

  for (int t = 1; t <= T; t++) {
    solve(t);
  }
}
```