Soluzioni — LUISSMatics 2019

Di seguito spiegheremo brevemente come risolvere i quattro task della LUISSMAtics 2019.

Esercizi in palestra (`palestra`) - molto facile

L'esercizio ci richiede di calcolare la massima massa muscolare che possiamo ottenere svolgendo determinati esercizi degli N proposti dal nostro allenatore.
Ogni esercizio ci permette di acquisire A_i massa muscolare.
Non avendo un buon allenatore questo valore può essere negativo, quindi per massimizzare la somma basta sommare solo gli esercizi positivi.
Complessità : $O(N)$
Una implementazione in C++ è la seguente:

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int T;
    cin >> T;
    for(int i = 1; i <= T; i++){
        int ans = 0;
        int N;
```

```
        cin >> N;
        for(int j = 0; j < N; j++){
            int x;
            cin >> x;
            ans += max(0,x);
        }
        cout << "Case #" << i << ": " << ans << '\n';
    }
    return 0;
}
```

Studenti assetati (`assetati`) - facile

IL Prof. Paperon De Paperoni ha dato il permesso ai suoi studenti di poter dissetarsi e questo arduo compito è stato assegnato proprio a noi. Per evitare di essere rimandati all' esame dobbiamo spendere il meno possibile!
Dobbiamo disetare N studenti avendo a disposizione R rubinetti e bicchieri con capacità C.
Questi rubinetti possono servire solo un tipo di bevanda e sono presenti M tipi diversi.
Ogni tipo di bevanda richiede D_i cl per dissetare una persona.
Nello specifico di ogni rubinetto sappiamo:

- Q_i cl rimanenti nel rubinetto
- B_i tipologia di bevanda che serve
- E_i costo per acquistare un bicchiere

Dovendo spendere il meno possibile è facile capire che la miglior strategia è di spendere il meno possibile per dissetare una singola persona ogni volta.
Per farlo non facciamo nient altro che provare a dissetare la singola persona con ogni rubinetto presente a patto che possegga abbastanza cl.
Il costo di un rubinetto è dato da quanti bicchieri sono neccessari per ottenere abbastanza cl del tipo di bevanda del rubinetto, per il suo costo.
Una volta trovato il rubinetto che ci fa spendere di meno aggiorniamo la quantità rimanente e ripetiamo il procedimento. Complessità : $O(N * R)$
Una implementazione in C++ è la seguente:

```
#include <bits/stdc++.h>
using namespace std;
int N,M,C,R;
#define MAXM 101
int D[MAXM];
#define MAXR 101
int Q[MAXR],B[MAXR],E[MAXR];
long long int solve(){
    long long int ans = 0;
    for(int i = 0; i < N; i++){
        int rubinetto = 0;
        int costo = INT_MAX;
        int qt = 0;
        for(int j = 0; j < R; j++){
```

```
            int need = D[B[j]];
            int paga = 0;
            int litri = 0;
            while(need > 0){
                need -= C;
                paga += E[j];
                litri += C;
            }
            if(Q[j] >= litri){
                if(paga < costo){
                    costo = paga;
                    rubinetto = j;
                    qt = litri;
                }
            }
        }
        Q[rubinetto] -= qt;
        ans += costo;
    }
    return ans;
}
int main(){
    int T;
    cin >> T;
    for(int i = 1; i <= T; i++){
        cin >> N >> C >> M >> R;
        for(int j = 0; j < M; j++){
            cin >> D[j];
```

```
        }
        for(int j = 0; j < R; j++){
            cin >> Q[j] >> B[j] >> E[j];
        }
        cout << "Case #" << i << ": " << solve() << endl <
< endl;
    }
    return 0;
}
```

Produzione di panini (`panini`) - Difficile

Dobbiamo guadagnare il più possibile per il bar della Luiss vendendo panini.
Per fare i panini abbiamo a disposizione N grammi di impasto e M tipi di ripieno.
Di ogni tipo di panino sappiamo:

- A_i grammi di ripieno disponibili
- B_i grammi di ripeno neccessari per prepararlo
- C_i grammi di impasto neccessari per perpararlo
- E_i guadagno in euro

Inoltre è disponibile il panino senza ripieno che richiede C_0 grammi di impasto e ci fa guadagnare E_0 euro.
Questo è un calssico problema di programmazione dinamica, dove dobbiamo massimizzare il guadagno.
La programmazione dinamica ci permette di ottenere la soluzione

migliore scomponendo il problema più grande in tanti sotto-problemi più piccoli facilmente risolvibili.
La difficoltà in questo genere di problemi è identificare l'istanza del sotto-problema (che chiamerò stato) che si sta calcolando e il come.
Le assunzioni svolgono un ruolo fondamentale, a seconda di esse possiamo provare una soluzione forza bruta o una soluzione con memoizzazione.
Nel nostro caso abbiamo: $N <= 1000$, $M <= 10$, $A_i <= 100$.
Una soluzione forza bruta in cui si provano tutte le combinazioni impiegherrebbe cosi tanto tempo che potremmo lasciare il pc accesso per l' eternità.
Quindi dobbiamo trovare un modo più intelligente di provarle tutte.
Qui entra in gioco la memoizzazione, genericamente è una tabella in cui si salvano i risultati intermedi.
Dobbiamo rappresentare lo stato di un problema in modo tale da non ricalcolarlo più volte.
Nel nostro caso è facilmente intuibile che uno dei discrimintatori degli stati è la quantità di impasto rimanente, ma ci basta per rappresentare ogni problema ? Ovvio che no, sapendo quanti grammi di impasto abbiamo usato non ci da altre informazioni riguardanti i ripieni rimanenti dei panini che abbiamo preparato fino ad addesso.
Se aggiungessimo nella nostra tabella la quantità di ripieno per ogni tipo di panino potrebbe funzionare.
Ma facendo due calcoli possiamo notare che non è una soluzione adeguata semplicmente per i limiti di memoria.
Dovremmo poter memorizzare *1000 * 10^10* interi che sono 'solo' 37252,902984619 GB di memoria.
Qua entra in gioco una piccola ottimizzazione fondamentale, la quantità

di ripieno di un determinato tipo di panino è indipendente dalle altre. Dando un ordine alla preparazione dei panini possiamo facilmente identificare uno stato in una tabellla $N * M$.
Potremo definrie la nostra programmazione dinamica come una semplice funzione ricorsiva che memorizza i risultati interemedi in cui in ogni stato prova a produrre tutti i panini che può in quella condizione salvando il massimo di ogni chiamata. La ricorsionse si interromperà nel caso in cui non si ha più impasto o siano finiti i tipi di panini.
Per semplificare la stesura del codice consideriamo il tipo di panino vuoto avente infinito ripieno.
Complessità : $O(N * M)$
Una implementazione in C++ è la seguente:

```
#include <bits/stdc++.h>
using namespace std;
int N,M;
#define MAXM 12
#define MAXN 10000
int memo[MAXN][MAXN];
int A[MAXM],B[MAXM],C[MAXM],D[MAXM];
int dp(int n,int m){
    if(n == 0 || m == M + 1)return 0;
    if(memo[n][m] != -1)return memo[n][m];
    int massa = 0, g = 0, ripieno = 0,ans = 0;
    while(massa <= n && ripieno <= A[m]){
        ans = max(ans,dp(n - massa,m + 1) + g);
        massa += C[m];
        g += D[m];
```

```
            ripieno += B[m];
        }
        memo[n][m] = ans;
        return ans;
}
int main(){
    int T;
    cin >> T;
    for(int i = 1; i <= T; i++){
        cin >> N >> M >> C[0] >> D[0];
        B[0] = 0;
        A[0] = INT_MAX;
        for(int j = 1; j <= M; j++){
            cin >> A[j] >> B[j] >> C[j] >> D[j];
        }
        memset(memo,-1,sizeof(memo));
        cout << "Case #" << i << ": " << dp(N,0) << '\n';
    }
    return 0;
}
```

Scambio culturale

Scambio culturale (Scambio culturale`) - Medio

Nel programma di scambio culturale della luiss si vogliono far

incontrare la coppia di docenti più lontana possibile in ambito scolastico e di diversa nazionalità.
Nello specifico bisonga calcolare la distanza massima tra un docente italiano e uno cinese in base alla categoria di studi.
La distanza viene calcolata come il numero di passi neccessari per raggiungere una determinata categoria partendo da un' altra.
Nella luiss sono presenti N docenti di cui sappiamo la loro nazionalità e categoria.
Ogni docente è descritto da una coppia in cui il primo valore descrive la nazionalità (0 se è italiano altrimenti 1 se è cinese) e la categoria di apparteneza.
Inoltre sono presenti M categorie più la generica che viene identificata dal numero 0.
L'associazione delle categorie identifica come si passa da una categoria ad un altra.
Per risolvere questo problema possiamo ricondurre questa associazioni a un grafo, più precisamente a un albero.
Un albero è un tipo particolare di grafo in cui sono presenti N nodi e $N - 1$ archi. Di conseguenza il grafo è connesso, non sono presenti cicli ed esiste un unico percorso per collegare due generci nodi A e B.
Per la risoluzione del problema possiamo avvalerci di questo grafo e tenere memorizzati due infomrazioni per ogni categoria. Se sono presenti docenti cinesi e inglesi. Una voltà fatto ciò possiamo risolvere il problema provando ogni combinaizione di categoria e per far ciò ci avvaliamo di una semplice dfs.
Nella dfs ci portiamo i parametri *Nodo, padre, ita,cina,dista* che identificano il nodo, il nodo da cui siamo partiti in modo tale da non tornare indietro, se il nodo di partenza principale aveva docenti italiani

e cinesi e la distanza con cui siamo giunti.

In questo modo a ogni chiamata possiamo controllare se nel nodo giunto possiamo creare la coppia e salvare la distanza massima.

Complessità : $O(M * M)$

Una implementazione in C++ è la seguente:

```
#include <bits/stdc++.h>
using namespace std;
int N,M;
#define MAXN 1000
#define MAXM 155
// 0 italiano 1 cinese
int tipo[MAXN][2];
vector <int> adj[MAXM];
int best = 0;
void dfs(int node,int p,bool ita,bool cina,int dista){
    if(cina && tipo[node][0]){
        best = max(dista,best);
    }
    if(ita && tipo[node][1]){
        best = max(dista,best);
    }
    for(int &i : adj[node]){
        if(i == p)continue;
        dfs(i,node,ita,cina,dista + 1);
    }
}
int solve(){
```

```
    best = 0;
    for(int i = 0; i <= M; i++){
        dfs(i,-1,tipo[i][0],tipo[i][1],0);
    }
    return best;
}
int main(){
    int T;
    cin >> T;
    for(int i = 1; i <= T; i++){
        memset(tipo,false,sizeof(tipo));
        for(int i = 0; i < MAXM; i++){
            adj[i].clear();
        }
        cin >> N >> M;
        for(int j = 0; j < N; j++){
            int cat,naz;
            cin >> naz >> cat;
            tipo[cat][naz] = true;
        }
        for(int i = 1; i <= M; i++){
            int p;
            cin >> p;
            adj[i].push_back(p);
            adj[p].push_back(i);
        }
        cout << "Case #" << i << ": " << solve() << '\n';
    }
```

```
    return 0;
}
```