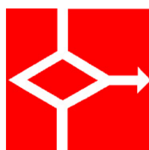
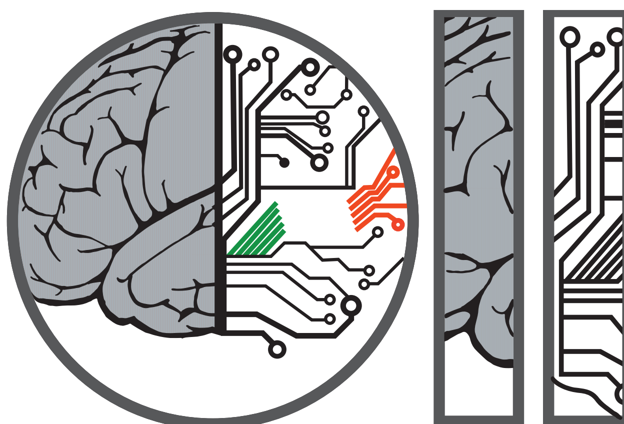
Ministero dell'Istruzione dell'Università e Ricerca



AICA
Associazione Italiana per l'Informatica
ed il Calcolo Automatico



OLIMPIADI ITALIANE DI
INFORMATICA
2015
Castiglione dei Pepoli

CASTIGLIONE DEI PEPOLI, 17 – 19 SETTEMBRE 2015

Finale nazionale

Testi e soluzioni ufficiali

Problemi a cura di

Luigi Laura

Coordinamento

Monica Gati

Testi dei problemi

Giorgio Audrito, Matteo Boscariol, William Di Luigi, Gabriele Farina, Gaspare Ferraro, Giada Franz, Federico Glaudo, Roberto Grossi, Luigi Laura, Dario Ostuni, Romeo Rizzi, Luca Versari

Soluzioni dei problemi

Giada Franz, Federico Glaudo

Supervisione a cura del Comitato per le Olimpiadi di Informatica

Indice

1 Torre di controllo (`aeroporto`) **1**

Testo del problema 1

Soluzione 4

2 Nemico mortale (`nemesi`) **6**

Testo del problema 6

Soluzione 9

3 Ordinamento a paletta (`paletta`) **12**

Testo del problema 12

Soluzione 14

Torre di controllo (aeroporto)

Limite di tempo: 0.2 secondi
Limite di memoria: 256 MiB

Nella torre di controllo dell'aeroporto di Bologna è scattato il panico. La finale nazionale delle Olimpiadi di Informatica, infatti, ha attirato una moltitudine di tifosi che stanno raggiungendo la città con voli di linea e charter. L'insolita mole di traffico sta quindi costringendo gli operatori della torre di controllo a regolare meticolosamente tutte le richieste di atterraggio, per evitare che la situazione porti a un rischio di incidente aereo troppo elevato.

In particolare, la torre di controllo ha ricevuto R richieste di atterraggio. L'i-esima richiesta arrivata in ordine cronologico consiste in due valori A[i] e B[i]: il primo di questi due valori indica l'istante di tempo in cui l'aereo corrispondente può atterrare se manovra direttamente verso la pista, mentre il secondo indica l'istante di tempo massimo entro cui può atterrare se cerca di allungare il tragitto (ma senza rischiare di finire il carburante).

La torre di controllo deve scegliere per ogni aereo un istante di tempo T[i] compreso tra A[i] e B[i], di modo che la distanza *minima* K tra due istanti scelti *sia massima possibile*. Infatti, più K si riduce e più aumenta il rischio di incidenti aerei. La torre di controllo deve anche considerare che un pilota vuole atterrare prima di tutti quelli che hanno fatto richiesta dopo di lui, perciò deve far sì che T[i] $<$ T[$i+1$], cioè che gli atterraggi avvengano nello stesso ordine delle richieste. Aiuta gli operatori a pianificare gli atterraggi!

Implementazione

Dovrai sottoporre esattamente un file con estensione `.c`, `.cpp` o `.pas`.

> ☞ Tra gli allegati a questo task troverai un template (`aeroporto.c`, `aeroporto.cpp`, `aeroporto.pas`) con un esempio di implementazione.

Dovrai implementare la seguente funzione:

■ Funzione `pianifica`

C/C++	`void pianifica(int R, int A[], int B[], int T[]);`
Pascal	`procedure pianifica(R: longint; A, B, T: array of longint);`

- L'intero R rappresenta il numero di richieste di atterraggio.
- Gli array A e B, indicizzati da 0 a $R-1$, contengono alla posizione i rispettivamente il minimo e il massimo istante di tempo possibile per l'atterraggio dell'aereo i.
- L'array T, indicizzato da 0 a $R-1$, dovrà essere riempito dal tuo programma con gli istanti di tempo T[i] compresi tra A[i] e B[i] tali da massimizzare la minima distanza tra due istanti scelti.

Il grader chiamerà la funzione `pianifica` e stamperà il valore calcolato in T sul file di output.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per testare le vostre soluzioni in locale. Il grader di esempio legge i dati di

input dal file `input.txt`, chiama le funzioni che dovete implementare e scrive il file `output.txt`, secondo il seguente formato.

Il file `input.txt` è composto da $R+1$ righe, contenenti:

- Riga 1: l'unico intero R.
- Righe $2, \dots, R+1$: i due interi $\texttt{A}[i]$, $\texttt{B}[i]$ per $i = 0, \dots, R-1$.

Il file `output.txt` è composto da un'unica riga, contenente:

- Riga 1: gli R interi $\texttt{T}[i]$ per $i = 0, \dots, R-1$ calcolati dalla funzione `pianifica`.

Assunzioni

- $1 \le R \le 100\,000$.
- $0 \le T_{\max} \le 1\,000\,000\,000$.
- $0 \le \texttt{A}[i] \le \texttt{B}[i] \le T_{\max}$ per ogni $i = 0, \dots, R-1$.
- È assicurato che in tutti i casi di prova sia possibile far atterrare gli aerei nello stesso ordine in cui sono giunte le richieste, in istanti di tempo distinti.

Assegnazione del punteggio

Il tuo programma verrà testato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo ad un subtask, è necessario risolvere correttamente tutti i test relativi ad esso.

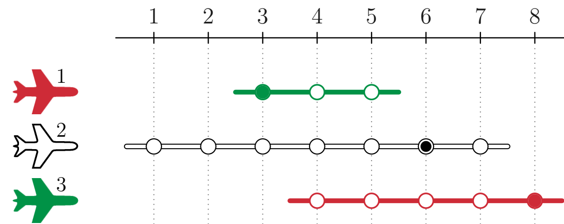
- **Subtask 1** [**5 punti**]: Casi d'esempio.
- **Subtask 2** [**7 punti**]: $\texttt{A}[i] = \texttt{B}[i]$ per ogni $i = 0, \dots, R-1$.
- **Subtask 3** [**14 punti**]: $T_{\max} \le 10R$.
- **Subtask 4** [**17 punti**]: $T_{\max} \le 1000$.
- **Subtask 5** [**15 punti**]: $T_{\max} \le 1\,000\,000$.
- **Subtask 6** [**20 punti**]: $\texttt{B}[i] - \texttt{A}[i] \le 100$.
- **Subtask 7** [**22 punti**]: Nessuna limitazione specifica.

Esempi di input/output

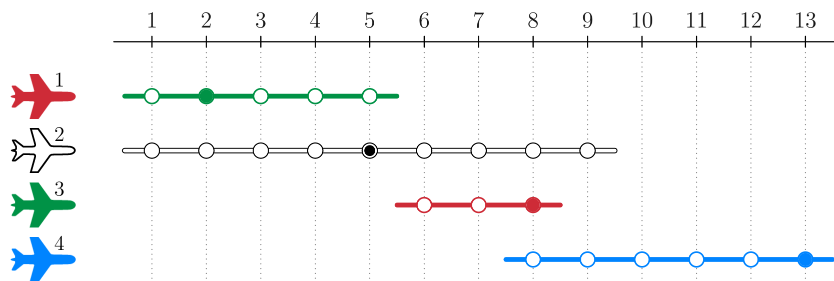
input.txt	output.txt
3 3 5 1 7 4 8	3 6 8
4 1 5 1 9 6 8 8 13	2 5 8 13

Spiegazione

Nel **primo caso di esempio**, si realizza $K = 2$ ed una possibile soluzione è:



Nel **secondo caso di esempio**, si realizza invece $K = 3$ ed una soluzione è:



Soluzione

La soluzione che prende punteggio pieno ha complessità $O(R\log(T))$, dove T è il massimo istante di tempo in cui sono compresi gli intervalli delle richieste.

Trovare la soluzione a K fissato

Dato un intero K, lo chiamiamo *buono* se è possibile far atterrare gli R aerei a distanza temporale almeno K uno dell'altro, *cattivo* altrimenti. Chiameremo inoltre $K_{\max}$ il massimo numero *buono*, che è quello ricercato dal problema.

Osserviamo innanzitutto che è facile verificare se un dato K è *buono* o *cattivo*, con un semplice algoritmo greedy. Sarà sufficiente, infatti, provare ad assegnare ordinatamente (cioè dall'aereo 0 all'aereo $R-1$) il primo tempo disponibile d'atterraggio. Questo significa che, assegnati i tempi di atterraggio per gli aerei da 0 a $i-1$, mi converrà provare a porre $\texttt{T}[i] = \min(\texttt{A}[i], \texttt{T}[i-1]+K)$.

Se in tale modo risulta esserci un i per cui $\texttt{T}[i]$ è maggiore di $\texttt{B}[i]$, capiamo che K è troppo grande per far atterrare gli aerei distanziati di K uno dall'altro. Altrimenti K è *buono* e abbiamo trovato dei tempi di atterraggio che distano almeno K uno dall'altro.

Soluzione con complessità $O(T)$

Vogliamo ora sfruttare quanto appena osservato per trovare una prima soluzione al problema.

Innanzitutto vale che $K_{\max}$ deve essere necessariamente compreso fra 1 e T/R, poiché se vogliamo che gli R interi $\texttt{T}[i]$ siano compresi fra 0 e T, ce ne saranno due che distano meno di T/R. Perciò per trovare $K_{\max}$, ci basta controllare $K = 1, \ldots, T/R$.

Quindi, tale soluzione, avrà complessità $O(R \cdot T/R) = O(T)$, poiché per verificare che un dato K funzioni il tempo è $O(R)$. In gara, già questa soluzione bastava per concludere il problema, ma in realtà non è necessario controllare tutti i K compresi fra 1 e T/R.

Soluzione con complessità $O(R\log(T/R))$

Manca un'unica osservazione conclusiva per abbassare il numero di K da controllare: dato un K *buono* necessariamente tutti i numeri minori di lui saranno ancora *buoni*; viceversa dato un K *cattivo*, anche tutti i numeri maggiori di lui saranno *cattivi*. Il punto chiave della soluzione sta quindi nell'usare una ricerca binaria.

Inizialmente abbiamo detto che $K_{\max}$ deve essere compreso fra 1 e T/R, poniamo quindi $l = 1$ ed $r = T/R$. Controlliamo ora se $K = m = (l+r)/2$ è *buono* o *cattivo*; nel primo caso è possibile restringere il nostro intervallo (in cui stiamo cercando $K_{\max}$) ponendo $l = m$, nel secondo caso ponendo $r = m - 1$.

Ci ritroviamo così con un nuovo intervallo su cui ripetere lo stesso ragionamento, finché nell'intervallo non rimane un unico intero (cioè $l = r$).

Ad ogni passaggio il nostro intervallo si dimezza, per cui il numero di passaggi che mi servono per arrivare ad un intervallo di lunghezza 1 è $\log(T/R)$.

In conclusione, tale soluzione ha quindi complessità $O(R\log(T/R))$.

Esempio di codice C++11

```
void pianifica(int R, int A[], int B[], int T[]) {
        // Ricerca binaria sul minor K possibile
        int l = 0;
        int r = 1e9;
        while (l+1  < r) {
                int m = (l+r)/2;
                bool buono = true;

                // Controllo se m risulta buono
                int FirstTimeFree = 0;
                for (int i = 0; i < R; i++) {
                        if (FirstTimeFree < A[i]) FirstTimeFree = A[i] + m;
                        else if (FirstTimeFree <= B[i]) FirstTimeFree += m;
                        else {
                                buono = false;
                                break;
                        }
                }

                if (buono) l = m;
                else r = m;
        }

        // Assumendo K=l, viene riempito adeguatamente l'array T[]
        int FirstTimeFree = 0;
        for (int i = 0; i < R; i++) {
                if (FirstTimeFree < A[i]) T[i] = A[i], FirstTimeFree = A[i] + l;
                else T[i] = FirstTimeFree, FirstTimeFree += l;
        }
}
```

Nemico mortale (nemesi)

Limite di tempo: 0.5 secondi
Limite di memoria: 256 MiB

Pochi sanno che Monica, oltre a gestire le Olimpiadi di Informatica, gestisce anche il "laboratorio creativo di pittura" della scuola elementare del suo quartiere. Anche quest'anno deve smistare i suoi fortunati alunni in gruppi di lavoro. Il compito è meno banale di quello che sembra: all'interno della classe, ogni bambino ha un nemico mortale (chi non lo aveva?). Inoltre, a complicare le cose, questa relazione non è necessariamente simmetrica: nonostante il nemico mortale di Luca sia Giacomo, il nemico mortale di Giacomo potrebbe non essere Luca.

Aiuta Monica a dividere la classe nel **minimo** numero di gruppi, in modo che all'interno di ogni gruppo non siano presenti contemporaneamente un bambino ed il suo nemico mortale.

Implementazione

Dovrai sottoporre esattamente un file con estensione `.c`, `.cpp` o `.pas`.

☞ Tra gli allegati a questo task troverai un template (`nemesi.c`, `nemesi.cpp`, `nemesi.pas`) con un esempio di implementazione.

Dovrai **implementare** la seguente funzione:

■ Funzione `smista`

C/C++	`void smista(int N, int nemico[]);`
Pascal	`procedure smista(N: longint; nemico: array of longint);`

- L'intero N rappresenta il numero di bambini nella classe di Monica. I bambini sono numerati con gli interi $0, 1, \ldots, N-1$.
- L'array `nemico` contiene, alla posizione i, il numero del nemico mortale del bambino i.

Il tuo programma potrà **utilizzare** le seguenti funzioni, definite nel grader:

■ Funzione `nuovo_gruppo`

C/C++	`void nuovo_gruppo();`
Pascal	`procedure nuovo_gruppo();`

Questa funzione crea un nuovo gruppo, al quale è possibile aggiungere bambini usando `aggiungi` (vedi sotto).

■ Funzione `aggiungi`

C/C++	`void aggiungi(int bambino);`
Pascal	`procedure aggiungi(int bambino);`

Questa funzione aggiunge il bambino numerato `bambino` all'ultimo gruppo creato.

Il grader chiamerà la funzione `smista`, la quale a sua volta assegnerà i bambini ai gruppi usando `nuovo_gruppo` e `aggiungi`.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per testare le vostre soluzioni in locale. Il grader di esempio legge i dati di input dal file `input.txt`, chiama la funzione che dovete implementare e scrive il file `output.txt`, secondo il seguente formato.

Il file `input.txt` è composto da due righe, contenenti:

- Riga 1: l'unico intero N.
- Riga 2: i valori `nemico[`i`]` per $i = 0, 1, \ldots, N-1$.

Il file `output.txt` è composto da G righe, dove G rappresenta il numero di gruppi creati dalla vostra soluzione. La riga i-esima contiene i numeri dei bambini assegnati al gruppo i, non necessariamente ordinati.

Assunzioni

- $2 \leq N \leq 100\,000$.
- Ogni bambino deve appartenere ad esattamente un gruppo.
- Nessun bambino è nemico mortale di se stesso.

Assegnazione del punteggio

Il tuo programma verrà testato su diversi test case raggruppati in subtask. Per ogni test case riceverai un punteggio di:

$$2^{\texttt{opt}-\texttt{eff}}$$

dove `opt` è il numero ottimale di gruppi e `eff` è il numero di gruppi ottenuto dalla tua soluzione. Per ogni subtask riceverai un punteggio pari al valore del subtask moltiplicato per il peggior punteggio ottenuto su uno dei suoi test case.

- **Subtask 1** [**5 punti**]: Casi d'esempio.
- **Subtask 2** [**13 punti**]: Le relazioni sono tutte simmetriche: se B è il nemico mortale di A, A è il nemico mortale di B.
- **Subtask 3** [**30 punti**]: Ogni bambino è nemico di al più un altro bambino.
- **Subtask 4** [**25 punti**]: $N \leq 2000$.
- **Subtask 5** [**27 punti**]: Nessuna limitazione specifica.

Esempi di input/output

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

Spiegazione

Nel **primo caso di esempio** è possibile dividere i bambini in due gruppi: il primo contiene $\{0, 2, 4\}$ e il secondo $\{1, 3\}$. Per costruire la suddivisione, sono state effettuate le chiamate: `nuovo_gruppo()`, `aggiungi(0)`, `aggiungi(4)`, `aggiungi(2)`, `nuovo_gruppo()`, `aggiungi(3)`, `aggiungi(1)`.

Nel **secondo caso di esempio** sono necessari tre gruppi: $\{1, 4\}$, $\{3\}$ e $\{0, 2\}$.

Soluzione

Per risolvere il problema sfrutteremo, come strumento principale, un'esplorazione dei grafi chiamata `dfs` o depth first search.

■ Costruzione del grafo associato al problema

Costruiamo innanzitutto un grafo non orientato i cui nodi sono i bambini. In questo grafo inseriamo un arco fra a e b se a è nemico mortale di b o b è nemico mortale di a; non ha importanza se è a a essere nemico di b o viceversa, dato che il risultato è lo stesso: a e b non possono stare nello stesso gruppo.

■ Risoluzione del problema in una componente connessa

Occupiamoci, per ora, di una singola componente connessa: vogliamo trovare il numero minimo di gruppi in cui i bambini ad essa appartenenti possono essere divisi. Equivalentemente, vogliamo colorare i nodi di questa componente con il minimo numero possibile di colori, in modo che due nodi dello stesso colore non siano mai collegati da un arco.

È bene dimostrare la seguente proprietà: se n è il numero di bambini nella componente connessa, allora il numero di archi è n oppure $n - 1$. Infatti:

- da un lato non possono esservi meno di $n - 1$ archi, altrimenti la componente non sarebbe connessa;
- dall'altro, ogni bambino ha esattamente un nemico mortale, e gli archi non possono essere più dei rapporti di inimicizia, che sono n.

Se la componente connessa che stiamo considerando ha un solo nodo, allora un colore è necessario e sufficiente; coloriamo quindi l'unico nodo presente di rosso.

Altrimenti prendiamo un nodo qualunque nella componente, coloriamolo di rosso e lanciamo una `DFS` partendo da quel nodo; questa `DFS` riempie un array `colore`, dove `colore`$[i]$ indica come vorremmo *temporaneamente* colorare il nodo i (usiamo 0 per rosso, 1 per blu). L'array `colore` viene riempito in modo che `colore`$[i]$ sia diverso da `colore` del padre di i. La `DFS` ha complessità

$$O((\text{numero di nodi}) + (\text{numero di archi})) = O(n + O(n)) = O(n)$$

Consideriamo ora l'albero generato dalla `DFS` (cioè l'insieme dei nodi e degli archi che la `DFS` ha attraversato). Se ci limitiamo a questo albero, possiamo notare che gli archi che appartengono ad esso non collegano mai due nodi dello stesso colore, per come abbiamo eseguito la `DFS`. Ma l'albero della `DFS` ha $n - 1$ archi (poiché ha n nodi), mentre la componente connessa intera, come abbiamo dimostrato in precedenza, ne ha n oppure $n - 1$.

Se effettivamente la componente connessa ha $n - 1$ archi, allora l'albero della `DFS` coincide con la componente connessa intera, quindi la colorazione che abbiamo trovato è valida (e ottimale: non è possibile colorare tutto con un solo colore).

Se invece la componente connessa ha n archi, allora $n - 1$ archi coincidono con quelli dell'albero della `DFS`, mentre uno è *in eccesso*.

- Se l'arco *in eccesso* collega due nodi di cui uno è rosso e l'altro blu, allora la colorazione è comunque valida (e ottimale).

- Se invece l'arco *in eccesso* collega due nodi dello stesso colore, la colorazione non è valida; ma possiamo risolvere il problema colorando uno dei due estremi dell'arco di verde. In questo caso 3 colori sono sufficienti, e dimostreremo ora che sono anche necessari.

  Siano a e b i due estremi dell'arco *in eccesso*. All'interno dell'albero della `DFS` la distanza fra a e b è pari (dato che i due nodi sono dello stesso colore); ma allora se partiamo da a, camminiamo fino a b seguendo l'albero della `DFS`, e poi percorriamo l'arco *in eccesso*, siamo tornati in a effettuando un numero dispari di passi; in altre parole, esiste un ciclo dispari all'interno della componente connessa.

  Supponiamo ora per assurdo che 2 colori siano sufficienti per colorare la componente connessa in modo valido; diciamo che il ciclo prima individuato ha lunghezza $2k + 1$ (con $k \geq 1$). Allora ci saranno almeno $k+1$ nodi nel ciclo con lo stesso colore (diciamo rosso senza perdita di generalità), e quindi almeno due nodi rossi saranno consecutivi nel ciclo, il che è assurdo.

  Dunque 3 colori sono necessari e sufficienti per colorare questa componente connessa.

Osserviamo che, per come abbiamo colorato i nodi nei vari casi, se c'è almeno un nodo verde c'è almeno un nodo blu, e se c'è almeno un nodo blu c'è almeno un nodo rosso.

■ Conclusione su tutto il grafo

Ripetiamo ora lo stesso procedimento su tutte le componenti connesse, in modo da colorare tutti i nodi del grafo.

La colorazione che otteniamo è valida: infatti, presi a e b dello stesso colore, se appartengono alla stessa componente connessa non sono collegati da un arco, poiché abbiamo colorato le singole componenti connesse in modo valido; se invece a e b appartengono a componenti connesse distinte, allora ovviamente non sono collegati.

La colorazione che otteniamo è anche ottimale: infatti utilizziamo tanti colori quanti ce ne sono nella componente connessa in cui ne abbiamo utilizzati di più, e in questa componente connessa la colorazione era ottimale.

A questo punto non rimane che inserire tutti i nodi rossi nel primo gruppo, i nodi blu (se ne esistono) nel secondo gruppo, i nodi verdi (se ne esistono) nel terzo gruppo. Quest'ultima operazione ha complessità $O(N)$.

Se chiamiamo $n_1, n_2, \ldots, n_c$ il numero di nodi nella prima, seconda,..., c-esima componente connessa (supponendo che le componenti connesse siano c), otteniamo che la complessità totale dell'algoritmo è

$$O(n_1) + O(n_2) + \ldots + O(n_c) + O(N) = O(N).$$

Esempio di codice C++11

Il sorgente che segue, pur usando una `dfs` sul medesimo grafo, si basa su un'idea lievemente diversa da quella esposta sopra: sfrutta la particolare struttura delle componenti connesse (che sono fondamentalmente un ciclo a cui sono appesi degli alberi) per dividere adeguatamente i bambini in gruppi.

```
1 #include <vector>
2 using namespace std;
3
4 void nuovo_gruppo();
5
```

```
6  void aggiungi(int bambino);
7
8  int numero_gruppi = 2; // I gruppi sono sempre almeno 2
9
10 vector<int> gruppi;
11 vector< vector<int> > graph;
12
13 bool dfs(int n, int g) {
14     if (gruppi[n] != -1)
15         return 1-gruppi[n] != g;
16     gruppi[n] = g;
17
18     bool ok = true;
19     for (auto v: graph[n]) {
20         ok &= dfs(v, 1-g);
21     }
22     if (!ok) { // Non bastano 2 gruppi
23         gruppi[n] = 2;
24         numero_gruppi = 3;
25     }
26     return true;
27 }
28
29 void smista(int N, int nemico[]) {
30
31     // Costruzione del grafo
32     graph.resize(N);
33     gruppi.resize(N, -1);
34     for (int i = 0; i < N; i++) {
35         graph[i].push_back(nemico[i]);
36         graph[nemico[i]].push_back(i);
37     }
38
39     // DFS di visita del grafo e assegnazione dei gruppi
40     for (int i = 0; i < N; i++)
41         if (gruppi[i] == -1)
42             dfs(i, 0);
43
44     // Costruzione dei gruppi tramite nuovo_gruppo() e aggiungi(i)
45     for (int j = 0; j < numero_gruppi; j++) {
46         nuovo_gruppo();
47         for (int i = 0; i < N; i++)
48             if (gruppi[i] == j)
49                 aggiungi(i);
50     }
51 }
```

Ordinamento a paletta (paletta)

Limite di tempo: 0.2 secondi
Limite di memoria: 256 MiB

Romeo ha di recente assistito ad un'avvincente performance di un cuoco acrobatico, che gestiva una imponente grigliata di braciole ribaltandole a gruppi di tre per mezzo di una apposita paletta. Questo evento gli ha ispirato l'idea del *paletta-sort*, una nuova interessante procedura di ordinamento.

Dato un vettore V contenente gli interi da 0 a $N-1$ (indicizzato da 0 a $N-1$), l'unica operazione ammessa nel paletta-sort è l'operazione *ribalta*. Questa operazione sostituisce tre elementi A, B, C consecutivi di V con i corrispondenti ribaltati C, B, A. Aiuta Romeo a capire se è possibile ordinare il vettore V, e in caso affermativo quante e quali operazioni *ribalta* sono sufficienti!

Implementazione

Dovrai sottoporre esattamente un file con estensione .c, .cpp o .pas.

> ☞ Tra gli allegati a questo task troverai un template (paletta.c, paletta.cpp, paletta.pas) con un esempio di implementazione.

Dovrai implementare la seguente funzione:

Funzione paletta_sort

C/C++	`long long paletta_sort(int N, int V[]);`
Pascal	`function paletta_sort(N: longint; V: array of longint) : int64;`

- L'intero N rappresenta il numero di elementi da ordinare.
- Il vettore V, indicizzato da 0 a $N-1$, contiene la sequenza da ordinare.
- La funzione dovrà restituire il numero ribaltamenti effettuati per ordinare V, oppure -1 se non c'è modo di ordinare il vettore.

Il grader chiamerà la funzione paletta_sort e ne stamperà il valore restituito sul file di output.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per testare le vostre soluzioni in locale. Il grader di esempio legge i dati di input dal file input.txt, chiama le funzioni che dovete implementare e scrive il file output.txt, secondo il seguente formato.

Il file input.txt è composto da due righe, contenenti:

- Riga 1: l'unico intero N.
- Riga 2: i valori V[i] per $i = 0, \ldots, N-1$.

Il file output.txt è composto da una riga, contenente:

- Riga 1: il valore R restituito dalla funzione paletta_sort.

Assunzioni

- $1 \leq N \leq 1\,500\,000$.
- $0 \leq \texttt{V}[i] \leq N - 1$ per ogni $i = 0, \ldots, N - 1$.

Assegnazione del punteggio

Il tuo programma verrà testato su diversi test case raggruppati in subtask. Per ogni test case riceverai punteggio:

- 1: se riporti correttamente il numero minimo di ribaltamenti;
- 0.2: se il vettore V è ordinabile e restituisci un numero maggiore o uguale a 0 (anche se non corrisponde al numero minimo di ribaltamenti), cioè riesci a distinguere quando il vettore è ordinabile;
- 0: altrimenti.

Per ogni subtask riceverai un punteggio pari al valore del subtask moltiplicato per il peggior punteggio ottenuto su uno dei suoi test case.

- **Subtask 1** [**5 punti**]: Casi d'esempio.
- **Subtask 2** [**19 punti**]: $N \leq 100$.
- **Subtask 3** [**24 punti**]: $N \leq 5000$.
- **Subtask 4** [**21 punti**]: $R \leq 100$ (oppure V non è ordinabile).
- **Subtask 5** [**25 punti**]: $N \leq 100\,000$.
- **Subtask 6** [**6 punti**]: Nessuna limitazione specifica.

Esempi di input/output

input.txt	output.txt
5 2 0 4 3 1	-1
6 2 3 0 5 4 1	3

Spiegazione

Nel **primo caso di esempio**, non è possibile ordinare il vettore dato.

Nel **secondo caso di esempio**, la soluzione proposta produce la seguente sequenza di ribaltamenti:

2	3	0	5	4	1

2	3	0	1	4	5

0	3	2	1	4	5

0	1	2	3	4	5

Soluzione

Per risolvere il problema bisogna accorgersi che le mosse di Romeo agiscono *indipendentemente* sulle caselle pari e dispari di `V`. Notato questo il problema si riduce a contare il numero di inversioni di una permutazione, che è un problema classico che si può risolvere con una struttura dati come il Fenwick tree oppure con un algoritmo analogo al *merge-sort* (in questo documento viene spiegata la soluzione tramite il *Fenwick tree*).

■ Spezzare il problema tra posti pari e dispari

Notiamo innanzitutto che le mosse di Romeo non cambiano la parità della posizione: in particolare i numeri che inizialmente stanno in $\{\texttt{V}[0], \texttt{V}[2], \texttt{V}[4], \dots\}$ rimarranno sempre in una posizione pari e altrettanto vale per quelli che si trovano in posizione dispari. Allora una facile condizione necessaria affinché si possa riordinare tramite il *paletta-sort* è che $\{\texttt{V}[0], \texttt{V}[2], \dots\}$ siano tutti pari e $\{\texttt{V}[1], \texttt{V}[3], \dots\}$ siano tutti dispari. Mostreremo che questa condizione è anche sufficiente.

Guardiamo come agiscono le mosse di Romeo su $\{\texttt{V}[0], \texttt{V}[2], \texttt{V}[4], \dots\}$: non fanno altro che swappare due elementi adiacenti. Perciò abbiamo ridotto il problema a contare quanti swap di elementi adiacenti sono necessari per ordinare un array (in realtà due array: sia $\{\texttt{V}[0], \texttt{V}[2], \dots\}$, sia $\{\texttt{V}[1], \texttt{V}[3], \dots\}$ e poi sommare le risposte).

■ Numero minimo di swap per ordinare un array

Immaginiamo di avere una permutazione $\texttt{A}[0], \texttt{A}[1], \dots, \texttt{A}[\texttt{N}-1]$ (ora l'N non è più quello del testo, ma uno generico) e di poter swappare due elementi adiacenti, cioè $\texttt{A}[\texttt{i}], \texttt{A}[\texttt{i}+1]$. Ci chiediamo quanti swap di elementi adiacenti siano necessari per ottenere $\texttt{A}[\texttt{i}] = \texttt{i}$.

Definiamo $I(\texttt{A})$ come il numero di coppie di indici $0 \leq i < j < N$ tali che $\texttt{A}[\texttt{i}] > \texttt{A}[\texttt{j}]$ e chiamiamolo il numero di inversioni della permutazione. È evidente che se `A` fosse ordinata allora varrebbe $I(\texttt{A}) = 0$ ed è vero anche il viceversa. Infatti se $I(\texttt{A}) = 0$ è facile dedurne che `A` è ordinata.

Inoltre è anche vero che un qualunque swap di elementi adiacenti fa variare $I(\texttt{A})$ esattamente di 1: in particolare se lo swap *ordina* i due elementi allora cala di 1 se invece li *inverte* aumenta di 1.

Unendo queste due osservazioni giungiamo a concludere che il minimo numero di swap necessari per ordinare `A` è proprio $I(\texttt{A})$. Infatti di meno non può essere perché all'inizio le inversioni sono $I(\texttt{A})$ e alla fine devono essere 0, ma ogni swap fa calare le inversioni al più di 1. Inoltre tale numero di swap è sufficiente: se `A` non è ordinata esiste sempre uno swap che "ordina" i due elementi adiacenti e perciò fa calare il numero di inversioni, quindi dopo $I(\texttt{A})$ swap (se adeguatamente scelti) si può avere che il numero di inversioni è 0 e cioè che la lista è ordinata.

■ Contare il numero di inversioni

Data una permutazione $\texttt{A}[0], \texttt{A}[1], \dots, \texttt{A}[\texttt{N}-1]$ vogliamo contare il numero di inversioni, cioè il numero di indici $i < j$ tali che $\texttt{A}[\texttt{i}] > \texttt{A}[\texttt{j}]$. Farlo in tempo $O(N^2)$ è ovvio, basta ciclare su tutte le coppie $0 \leq i < j < N$ ed incrementare il risultato se $\texttt{A}[\texttt{i}] > \texttt{A}[\texttt{j}]$. Un modo alternativo, sempre $O(N^2)$, per ottenere il risultato è ciclare su $0 \leq i < N$ e per ogni i aumentare il risultato di k_i dove k_i è il numero di indici $j > i$ tali che $\texttt{A}[\texttt{j}] < \texttt{A}[\texttt{i}]$. Noi troveremo un modo per implementare quest'ultimo algoritmo in $O(N \log(N))$ sfruttando la struttura dati chiamata *Fenwick tree* (in realtà va anche bene un *range tree*, ma per prendere l'ultimo subtask serve un *Fenwick* perché nella pratica è più veloce di un fattore costante).

In particolare inizializziamo un *Fenwick tree* in modo che gestisca due operazioni sull'array `val[0..N]`: dobbiamo poter chiedere che valore c'è in una casella e incrementare di 1 tutte le caselle con indice da 0 ad x.

Poniamo `res` $= 0$ e cicliamo su $0 \leq i < N$ (in ordine crescente). Inizialmente `val[x]` $= 0$ per ogni $0 \leq x < N$ e, al passo $i = k$, `val[x]` rappresenterà quanti numeri *minori* di k sono presenti nella permutazione dopo il posto x, cioè gli indici $j > x$ tali che `val[j]` $<$ `k`. Allora al passo i, troviamo s_i tale che `A[s_i]` $=$ `i` (con un preprocessing lineare questo si può fare in tempo costante) e sommiamo a `res` il valore `val[s_i]`, cioè proprio il numero di $j > s_i$ tali che `A[j]` $<$ `i` (quindi k_{s_i}). A questo punto aggiorniamo `val[0..N]` incrementando di 1 tutte le caselle dalla 0 alla s_i.

Alla fine del ciclo il valore di `res` sarà proprio il numero di inversioni della permutazione.

Esempio di codice C++11

```
#define MAXN 1500000

int N;
int A[MAXN], ord[MAXN], fen[MAXN];

int ask(int i) {
        int res = 0;
        for (; i >= 0; i = (i&(i+1))-1) res += fen[i];
        return res;
}

void add(int i, const int N) {
        for (; i < N; i = i|(i+1)) fen[i]++;
}

long long Solve(int N) {
        for (int i = 0; i < N; i++) fen[i] = 0;
        for (int i = 0; i < N; i++) ord[A[i]] = i;
        long long res = 0;
        for (int i = N-1; i >= 0; i--) {
                res += (long long)ask(ord[i]);
                add(ord[i], N);
        }
        return res;
}

long long paletta_sort(int N, int V[]) {
        // Controllo che sia possibile ordinare gli elementi
        for (int i = 0; i < N; i++) {
                if (V[i]%2 != i%2) return -1;
        }

        // Contare gli scambi sui due sotto-array (pari e dispari)
        long long res = 0;

        for (int i = 0; i < N; i+=2) A[i/2] = V[i]/2;
        res += Solve((N+1)/2);

        for (int i = 1; i < N; i+=2) A[(i-1)/2] = (V[i]-1)/2;
        res += Solve(N/2);

        return res;
}
```