ABC 2019

Testi e soluzioni dei problemi

Innumerevoli progetti (`progetti`)

Leonardo è universalmente riconosciuto come uno degli inventori più prolifici di tutti i tempi. Com'era possibile che non si dimenticasse le numerose idee che gli balenavano continuamente in testa? Probabilmente Leonardo aveva cura per ciascuna idea di appuntarsi qualche bozza su un foglietto, in modo che non appena fosse stato possibile sarebbe stato in grado di recuperare le idee e portare a termine il progetto.

Figura 1: Si crede che Leonardo sia stato tra i primi a costruirsi e usare una penna stilografica.

Luca è strabiliato dall'impressionante numero di progetti e fatica a capacitarsi di come una singola persona abbia potuto lavorare a tutto ciò. È ragionevole supporre infatti che il massimo numero di progetti che si possono realizzare in un giorno sia un numero K piccolo. Inoltre, non tutti i giorni sono uguali: a volte si riesce a completarne solo uno... a volte molti!

Luca sta provando a stimare il numero di giorni necessari per completare N progetti ma è in confusione perché si è accorto che il risultato varia estremamente a seconda della quantità di progetti completati in ciascun giorno. Facendo un passo indietro, vuole allora capire un'altra cosa: quante sono le possibili combinazioni diverse con cui può completare tutti i progetti, scegliendo arbitrariamente quanti progetti completare giorno per giorno e senza avere alcun limite massimo di giorni?

Implementazione

Dovrai sottoporre un unico file con estensione `.cpp` o `.c`.

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

Dovrai implementare la seguente funzione:

■ **Funzione** `progetti`

C/C++	`int progetti(int N, int K);`

- L'intero N rappresenta il numero dei progetti.
- L'intero K rappresenta il massimo numero di progetti completabili in un giorno.
- La funzione dovrà restituire il numero di modi diversi con cui è possibile completare i progetti. Poiché tale numero può essere grande, è necessario restituire soltanto il resto della divisione (operatore di *modulo*[1]) per 1 000 000 007.

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

Allegata 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 da `stdin`, chiama la funzione che dovete implementare e scrive su `stdout`, secondo il seguente formato.

Il file di input è composto da un'unica riga, contentente:

- Riga 1: gli interi N e K, separati da uno spazio.

Il file di output è composto da un'unica riga, contenente:

- Riga 1: il valore restituito dalla funzione `progetti`.

Assunzioni

- $2 \leq N \leq 10^9$.
- $2 \leq K \leq 5$.
- In ogni giorno Leonardo può completare da 1 a K progetti.
- Rigorosmente, due modi di completamento si considerano *diversi* se differiscono per il numero di giorni necessari al completamento di tutti i progetti oppure se, a parità di giorni, esiste almeno un giorno in cui nei due modi sono stati completati un numero diverso di progetti.

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** [**0 punti**]: Casi d'esempio.
- **Subtask 2** [**15 punti**]: $K = 2$, $N \leq 15$.
- **Subtask 3** [**20 punti**]: $K = 2$, $N \leq 10^6$.
- **Subtask 4** [**15 punti**]: $K = 3$, $N \leq 10^6$.
- **Subtask 5** [**20 punti**]: $K = 2$, $N \leq 10^9$.
- **Subtask 6** [**30 punti**]: Nessuna limitazione specifica.

[1] In C/C++ l'operatore modulo ha simbolo '`%`'.

Esempi di input/output

input	output
2 2	2
4 2	5

Spiegazione

Nel **primo caso di esempio** ci sono due modi diversi per completare i due progetti rispettando il vincolo dei due progetti al giorno: è possibile completarne uno nel primo giorno e uno nel secondo oppure si può finire in un solo giorno completandoli subito entrambi.

Nel **secondo caso di esempio** ci sono cinque modi diversi per completare i quattro progetti:

- uno al giorno per quattro giorni;
- due al giorno per due giorni;
- uno il primo giorno, due il secondo giorno, uno il terzo giorno;
- uno il primo giorno, uno il secondo giorno, due il terzo giorno;
- due il primo giorno, uno il secondo giorno, uno il terzo giorno.

Soluzione

Questo problema ammette (facili) soluzioni specifiche per alcuni subtask con $K = 2$; nella trattazione che segue ci concentriamo invece sulla versione più generale con K arbitrario.

■ Soluzione esponenziale $\mathcal{O}(K^N)$

Questo problema si presta bene a una soluzione *ricorsiva*. Definiamo una comoda funzione $f_k(n)$ che conta il numero di combinazioni possibili con cui è possibile completare n progetti svolgendone *al più k* al giorno. Si ha che:

- $f_k(1) = 1$ indipendentemente da k: infatti, a prescindere da quanti progetti possiamo completare in un giorno, vi è un unico modo per completare l'unico progetto rimasto;
- $f_k(n) = 0$ per ogni $n \leq 0$ indipendentemente da k: infatti, se non vi è alcun progetto da completare, non c'è alcun modo di fare ciò.

Se in un certo giorno Leonardo può completare al più k progetti e ne ha n da fare, può farne uno (gliene restano $n - 1$), oppure può farne due (gliene restano $n - 2$) e così via fino a completarne k (il massimo consentito, gliene restano $n - k$). Questi sono tutti modi diversi per completare il lavoro a prescindere da cosa farà nei giorni successivi, pertanto vanno sommati. Possiamo esprimere in forma compatta il caso generale quindi come

$$f_k(n) = \sum_{x=1}^{k} f_k(n - x)$$

La risposta al problema è ovviamente $f_K(N)$. A questo punto è facile implementare la funzione ricorsiva descritta, che ha complessità esponenziale.

■ Soluzione $\mathcal{O}(NK)$ tramite programmazione dinamica

Ci si accorge presto che la soluzione esponenziale è inefficiente, in quanto calcola più volte la funzione f con gli stessi parametri. Mantenendo invariata la definizione di $f_k(n)$, possiamo sfruttare la tecnica della programmazione dinamica per memorizzare il risultato della funzione a seconda dei parametri.

Ci sono N possibilità per l'argomento della funzione e il calcolo di ciascuna può richiedere sommare fino a K addendi; questo pone un limite superiore pari a $\mathcal{O}(NK)$ alla complessità computazionale di questa soluzione. Poiché N può essere molto grande ($N \leq 10^9$), è necessario conservare in memoria solo le combinazioni $f_k(n)$ effettivamente richieste nei calcoli e non preallocare una grossa area di memoria, decisamente troppo grande per un computer!

■ Soluzione $\mathcal{O}(K^3 \log N)$ tramite esponenziazione veloce di matrici

Analizzando attentamente quanto viene richiesto, si nota che la risposta a questo problema è data dall'n-esimo elemento di una ricorrenza lineare.

Prodotto tra matrici

Il prodotto tra matrici bidimensionali richiede che esse siano *conformabili*, ovvero che il numero di colonne della prima matrice corrisponda al numero di righe della seconda. In questa situazione, la matrice prodotto ha lo stesso numero di righe della prima e lo stesso numero di colonne della seconda.
Detta $A^{x \times y}$ la prima matrice e $B^{y \times z}$ la seconda, la matrice prodotto avrà forma $C^{x \times z}$ e i suoi elementi saranno dati da:

$$C[i][j] = \sum_{k=0}^{y-1} A[i][k] \cdot B[k][j]$$

Come si può intuire dalla formula, calcolare il prodotto di tali matrici ha complessità $\mathcal{O}(x \cdot y \cdot z)$.

Per comodità e semplicità ci limitiamo a spiegare l'approccio con $K = 3$, ma il discorso si generalizza facilmente al variare di K.

Posto $K = 3$, la ricorrenza lineare F è definita come:

$$F_n = F_{n-1} + F_{n-2} + F_{n-3}$$

Il calcolo dell'$(n + 1)$-esimo elemento dati quelli precedenti può essere effettuato tramite prodotto di opportune matrici (tecnica chiamata *matrix exponentiation*). Nello specifico, date

$$A = \begin{pmatrix} \mathrm{F}_n \\ \mathrm{F}_{n-1} \\ \mathrm{F}_{n-2} \end{pmatrix} \qquad B = \begin{pmatrix} \mathrm{F}_{n+1} \\ \mathrm{F}_n \\ \mathrm{F}_{n-1} \end{pmatrix}$$

vorremmo trovare una matrice C tale che $C \cdot A = B$. Ragionando sulla definizione del prodotto tra matrici descritta sopra, si può calcolare che posto

$$C = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

l'espressione $C \cdot A = B$ risulta corretta. Abbiamo quindi un modo per calcolare l'$(n+1)$-esimo elemento della sequenza dati gli elementi nelle posizioni n, $n - 1$ e $n - 2$.

La risposta al problema sarà quindi contenuta nella posizione $[0, 0]$ della matrice C elevata alla $(N - 1)$-esima potenza (tale cella contiene il valore di F_{n+1}, quindi di F_N nel caso specifico).

Dobbiamo prestare attenzione però a *come* calcoliamo la matrice C^{N-1}. Si potrebbe essere tentati di calcolare $C^2, C^3, \ldots, C^{N-1}$ ma così facendo sarebbero necessari $N - 2$ prodotti tra matrici $K \times K$ e ciascuno di essi richiede $\mathcal{O}(K^3)$. Esiste un modo più furbo per ottenere lo stesso risultato che impiega una tecnica detta *fast exponentiation*.

Per calcolare a^b (siano essi numeri reali o matrici) possiamo osservare che

$$a^b = \begin{cases} (a^2)^{b/2} & \text{se } b \text{ è pari} \\ a \cdot (a^2)^{(b-1)/2} & \text{se } b \text{ è dispari} \end{cases}$$

Calcolare potenze in questo modo richiede solo $\mathcal{O}(\log b)$ passi, un miglioramento significativo rispetto all'algoritmo naïve lineare. Per dettagli in merito all'implementazione si può consultare la soluzione pubblicata sul progetto GitHub della gara[2] oppure cercare informazioni in merito a *exponentiation by squaring*, un altro nome con cui è nota questa tecnica.

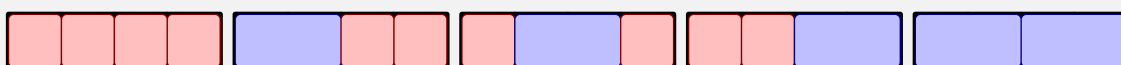
[2] `https://github.com/lucach/abc`

Riassumendo, possiamo calcolare C^{N-1} in $\mathcal{O}(K^3 \log N)$ ed ottenere il risultato finale nella cella [0][0] della matrice trovata.

■ Soluzione $\mathcal{O}(K^3 \log N + K \log^2 N)$ tramite calcolo combinatorio

Presentiamo ora una soluzione alternativa efficiente, ma di cui non dimostreremo la complessità computazionale. Essa segue un'idea interessante e non implementata da nessun partecipante durante la gara, sfruttando la seguente analogia: l'obiettivo è tassellare un rettangolo $1 \times n$ utilizzando tasselli di dimensioni $1 \times 1, 1 \times 2, \ldots, 1 \times k$. Ciò premesso, il valore della funzione $f_k(n)$, definita come in precedenza, è dato dal numero di modi con cui riusciamo a tassellare il rettangolo $1 \times n$ utilizzando un numero arbitrario di tasselli $1 \times 1, 1 \times 2, \ldots, 1 \times k$. In questo parallelismo, le varie dimensioni dei tasselli rappresentano quindi il numero di progetti completati in un giorno.

Per chiarezza, mostriamo questo approccio con un esempio dove $N = 4$ e $K = 2$; la soluzione è $f_2(4) = 5$. Infatti, i 5 modi che abbiamo per tassellare il rettangolo 1×4 avendo a disposizione tasselli 1×1 e 1×2 sono:



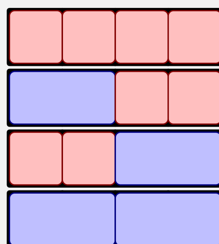
Ogni tassello $1 \times x$ indica che in quel giorno Leonardo ha svolto x progetti.

Il "tempo" scorre da sinistra a destra (quindi l'ordine conta!)

Definiamo i casi base della funzione in modo identico a quanto fatto in precedenza ($f_k(1) = 1$ e $f_k(x) = 0$ per $x < 0$). Per questione di comodità che sarà chiara in seguito (1 è l'elemento neutro della moltiplicazione), poniamo anche $f_k(0) = 1$.

Per ottenere il numero totale di combinazioni, iniziamo contando il numero di combinazioni che otteniamo unendo 2 rettangoli larghi la metà di quello finale, quindi $f_k(\lfloor n/2 \rfloor) \cdot f_k(\lceil n/2 \rceil)$.

Nel caso di esempio queste combinazioni sono $f_k(2) \cdot f_k(2) = 4$, ovvero:



Come si nota bene nella rappresentazione grafica, tutte queste combinazioni richiedono che il secondo ed il terzo progetto siano eseguiti in giorni differenti (lo si nota dal fatto che non capita mai che un tassello sia "a cavallo" della metà del rettangolo, nell'esempio quindi tra il secondo e il terzo progetto).

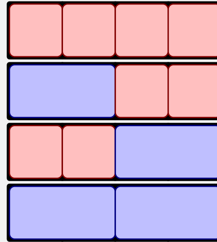
Dobbiamo quindi aggiungere alla quantità calcolata anche le combinazioni che prevedono che il secondo ed il terzo progetto siano svolti nello stesso giorno (graficamente è l'equivalente di provare ad inserire ogni possibile tassello di dimensione *maggiore* di 1 e al più lungo K in ogni possibile modo per il quale risulti che il secondo ed il terzo progetto siano svolti nello stesso giorno).

In questo caso specifico l'unico modo è inserire un tassello 1×2 esattamente al centro del rettangolo 1×4; le combinazioni con cui tassellare i due rettangoli rimanenti (prima e dopo il tassello inserito al centro) possono essere calcolate nuovamente tramite la funzione f_k.

Nel caso specifico, dopo aver inserito un tassello 1×2 al centro restano un progetto prima (copribile in $f_2(1) = 1$ modo) e un progetto dopo (copribile in $f_2(1) = 1$ modo). In conclusione avremo quindi $1 \cdot 1 = 1$ modo aggiuntivo per tassellare il rettangolo, portando il totale a $4 + 1 = 5$.

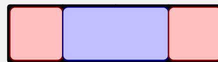
Un secondo esempio leggermente più elaborato

Mostriamo ora il ragionamento su un esempio leggermente più grande per rendere ancora meglio l'idea con $N = 4$ e $K = 3$. Come abbiamo fatto in precedenza, consideriamo per iniziare le combinazioni ottenibili tassellando le due metà 1×2. Avremo $f_3(\lfloor n/2 \rfloor) \cdot f_3(\lceil n/2 \rceil) = f_3(2) \cdot f_3(2) = 2 \cdot 2 = 4$. Le quattro combinazioni sono infatti identiche alle precedenti (sui rettangoli 1×2 infatti il tassello 1×3 è inutilizzabile):



Aggiungiamo ora le combinanzioni rimanenti, ricordandoci che ora possiamo utilizzare anche tasselli di larghezza 3:

- Il secondo ed il terzo progetto sono svolti nel secondo giorno (tassello azzurro). I rimanenti rettangoli possono essere tassellati in $f_3(1) \cdot f_3(1) = 1 \cdot 1 = 1$ modo.



- Il secondo ed il terzo progetto sono svolti nel primo giorno insieme al primo (tassello verde). Resta solo un rettangolo a destra, che può essere tassellato in $f_3(0) \cdot f_3(1) = 1 \cdot 1 = 1$ modo.

- Il secondo ed il terzo progetto sono svolti nel secondo giorno insieme al quarto (tassello verde). Resta solo un rettangolo a sinistra, che può essere tassellato in $f_3(1) \cdot f_3(0) = 1 \cdot 1 = 1$ modo.

Abbiamo quindi che il valore cercato di $f_3(4)$ è pari a $4 + 1 + 1 + 1 = 7$.

Esempio di codice C++11

```
#include<bits/stdc++.h>

std::unordered_map<int,int> M;
const int MOD = 1000000007;
int K;

int comb(int n) {
    if (n < 0)
        return 0;
    if (M[n])
        return M[n];
    int meta_sx = n/2, meta_dx = (n+1)/2;
    int risposta = ((long long)comb(meta_sx) * comb(meta_dx)) % MOD;
    for (int lunghezza = 2; lunghezza <= K; lunghezza++)
        for (int offset = 1; offset < lunghezza; offset++) {
            long long altre = (long long)comb(meta_sx - offset) *
                                 comb(meta_dx - lunghezza + offset);
            risposta = (risposta + altre) % MOD;
        }
    M[n] = risposta;
    return risposta;
}
int progetti(int N, int K) {
    ::K = K;
    M[0] = 1;
    M[1] = 1;
    return comb(N);
}
```

Lotteria di quadri (`quadri`)

Facendo pulizia in soffitta Fabio ha ritrovato diversi quadri, alcuni del tutto privi di valori e altri invece potenzialmente interessanti (tra cui diverse copie di alcuni dipinti di Leonardo da Vinci). Per racimolare qualche soldo, ha intenzione di venderli: ha pertanto portato a valutare gli N quadri da un esperto che ha attribuito a ciascuno un valore V_i approssimativo.

Per far sì che tutti i quadri di minor valore non restino invenduti, continunando a ingombrare la soffitta, Fabio è intenzionato a organizzare una lotteria con una regola particolare: le opere saranno disposte in fila e l'acquirente, dopo aver pagato un "biglietto di ingresso", potrà scegliere a propria discrezione un "blocco" consecutivo di B opere (non di più né di meno).

Figura 1: Alcuni dei quadri esposti (foto di Muhammad Raufan Yusup).

Fabio non vuole che la somma dei valori delle opere che un acquirente può portarsi a casa sia maggiore di un certo massimale M, altrimenti ci starebbe perdendo troppo. D'altro canto, pur rispettando questo principio, vorrebbe rendere B (il numero di quadri che ci si porta a casa comprando il biglietto d'ingresso) più alto possibile. Aiutalo a capire qual è il valore massimo di B per rendere il più appetibile possibile la lotteria!

Implementazione

Dovrai sottoporre un unico file con estensione `.cpp` o `.c`.

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

Dovrai implementare la seguente funzione:

■ **Funzione `quadri`**

C/C++	`int quadri(int N, long long M, int V[]);`

- L'intero N rappresenta il numero dei quadri.
- L'intero M rappresenta il massimale da non superare.
- L'array `V`, indicizzato da 0 a $N-1$, contiene alla posizione i il valore dell'i-esimo quadro nella fila.
- La funzione dovrà restituire il valore massimo per B come descritto nel testo.

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

Allegata 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 da `stdin`, chiama la funzione che dovete implementare e scrive su `stdout`, secondo il seguente formato.

Il file di input è composto da due righe, contenenti:

- Riga 1: gli interi N e M, separati da uno spazio.
- Riga 2: N interi `V`[i] per $i = 0, \ldots, N-1$.

Il file di output è composto da un'unica riga, contenente:

- Riga 1: il valore restituito dalla funzione `quadri`.

Assunzioni

- $1 \leq N \leq 200\,000$.
- $1 \leq M \leq 10^{12}$.
- $1 \leq V_i \leq 10^6$ per ogni $i = 0 \ldots N-1$.
- Nella scelta del blocco di B quadri, l'acquirente **non** può scegliere un blocco agli estremi della fila in modo da prenderne meno di B.

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** [**0 punti**]: Casi d'esempio.
- **Subtask 2** [**15 punti**]: $M < V_i$ per ogni $i = 0, \ldots, N-1$ oppure $M > V_0 + V_1 + \ldots + V_{N-1}$.
- **Subtask 3** [**20 punti**]: $N \leq 500$.
- **Subtask 4** [**25 punti**]: $N \leq 5\,000$.
- **Subtask 5** [**40 punti**]: Nessuna limitazione specifica.

Esempi di input/output

input	output
4 8 1 2 3 4	2
5 1 3 1 3 2 10	0

Spiegazione

Nel **primo caso di esempio** è possibile impostare $B = 2$: l'acquirente potrebbe scegliere i quadri di valore $1 + 2 = 3$, $2 + 3 = 5$ oppure $3 + 4 = 7$ senza superare mai il massimale $M = 8$. Non sarebbe stato possibile scegliere $B = 3$ perché in quel caso l'acquirente avrebbe potuto il scegliere il blocco di valore $2 + 3 + 4 = 9$, superando il massimale.

Nel **secondo caso di esempio** anche scegliere $B = 1$ consentirebbe all'acquirente di superare il massimale qualunque scelta egli faccia (ad eccezione del secondo quadro). La risposta corretta è quindi $B = 0$.

Soluzione

■ Soluzione naïve $\mathcal{O}(N^3)$

Si può cercare il valore massimo di B semplicemente implementando quanto descritto nel testo del problema. Si prova con $B = 1$, $B = 2$ e così via, incrementando il valore ad ogni tentativo, fino a trovare il primo valore per cui esiste un sottoarray lungo B la cui somma dei valori supera M. Per verificare ciò, fissato il valore di B è necessario controllare tutti gli intervalli $[i, i + B)$ con $i = 0 \ldots N - 1 - B$ (ovvero tutte le possibili posizioni di partenza), calcolarne la somma e verificare che sia $\leq M$. Non appena si trova un intervallo con somma maggiore di M, possiamo terminare gli incrementi di B e affermare che la risposta è $B - 1$.

Questa soluzione ha complessità $\mathcal{O}(N^3)$ in quanto per ogni possibile valore di B (al più N incrementi, da 1 a N) e per ogni possibile posizione di partenza di un intervallo (che sono $N - B$, quindi $\sim N$) sommiamo B valori.

■ Soluzione $\mathcal{O}(N^2)$

Modificando l'approccio naïve è possibile ridurre il tempo impiegato per controllare se un valore di B è adatto oppure no da $\mathcal{O}(N^2)$ a $\mathcal{O}(N)$.

Nello specifico l'inefficienza è data dal ricalcolo completo della somma degli elementi di due intervalli aventi inizio rispettivamente nella posizione i e nella posizione $i + 1$. È facile accorgersi che questi due intervalli $[i, i + B)$ e $[i + 1, i + 1 + B)$ differiscono solo per l'elemento in posizione i (che c'è nel primo intervallo ma non nel secondo) e per quello in posizione $i + B$ (che c'è nel secondo intervallo ma non nel primo). Con questa semplice tecnica, che è nota come "sliding window", la somma del prossimo intervallo può essere calcolata in tempo costante una volta nota la somma dell'intervallo corrente, aggiungendo e togliendo i valori dei due elementi menzionati sopra.

Questo accorgimento riduce il tempo complessivo da $\mathcal{O}(N) \cdot \mathcal{O}(N^2)$ a $\mathcal{O}(N) \cdot \mathcal{O}(N) = \mathcal{O}(N^2)$.

■ Soluzione $\mathcal{O}(N \log N)$

Ora che sappiamo come verificare in tempo lineare se un certo B è ammissibile oppure no, possiamo pensare di diminuire il numero di valori da controllare.

È possibile utilizzare la ricerca dicotomica in quanto sono verificate le seguenti due osservazioni, grazie al fatto che non vi sono valori negativi:

- Se la somma dei valori in tutti gli intervalli lunghi X non supera M, allora anche tutti gli intervalli più corti avranno somma non superiore a M.
- Se la somma dei valori in almeno un intervallo lungo X supera M, allora per tutte le lunghezze maggiori sicuramente esisterà almeno un intervallo di quella lunghezza la cui somma supera M.

Con la ricerca dicotomica vengono verificati, ciascuno in tempo lineare, al più $\log N$ valori per B, portando a una complessità finale di $\mathcal{O}(N \log N)$, che è sufficiente per ottenere il punteggio pieno.

■ Soluzione $\mathcal{O}(N)$

Per ottenere una soluzione lineare in N dobbiamo cambiare il ragionamento precedente: anziché verificare se un certo valore di B è ammissibile, è possibile per ogni posizione di partenza controllare quanto è lungo il sottoarray massimo con somma non superiore a M. Tra tutte queste lunghezze, la risposta al problema è il valore minimo (visto che, fissato B, *tutti* gli intervalli devono rispettare la condizione).

Quindi, più precisamente, per ogni posizione i di inizio cerchiamo la posizione j tale che la somma dell'intervallo $[i, j]$ non superi M mentre la somma dell'intervallo $[i, j + 1]$ superi M. In altre parole, stiamo dicendo che non è possibile spostare j nemmeno di una posizione in avanti senza superare M. La

lunghezza di questi intervalli è pari a $j - i + 1$ e, come detto sopra, tra tutte queste lunghezze scegliamo la più corta.

Si nota abbastanza facilmente che man mano che i viene incrementato, j non decresce mai (ricordiamo ancora che i valori sono tutti positivi); tali contatori verranno incrementati quindi entrambi al più N volte, mantenendo così complessità totale $\mathcal{O}(N)$.

Esempio di codice C++11

```
1   #include <algorithm>
2   int quadri(int N, long long M, int V[]) {
3       int risposta = N;
4       long long window = 0;
5       for(int start = 0, end = 0; end < N; start++) {
6           while (end < N and window + V[end] <= M) {
7               window += V[end];
8               end++;
9           }
10          risposta = std::min(risposta, end-start);
11          window -= V[start];
12      }
13      return risposta;
14  }
```

Cenacolo (`ultimacena`)

Tra le varie opere di Leonardo, l'affresco raffigurante l'Ultima Cena è considerato uno dei maggiori capolavori. Un amico di Luca sta svolgendo una ricerca approfondita con il compito di fornire una descrizione esaustiva e particolareggiata di queste meraviglie.

Analizzando l'affresco, ha già individuato N "zone di interesse" (come quelle racchiuse tra ellissi nella figura sotto), descritte con interi da 1 a N, e M relazioni tra di esse (indicate in figura con delle linee). I motivi per definire una relazione tra due zone sono molti: potrebbero contenere due personaggi che si stanno guardando, oppure due aree dipinte con tecniche similiari oppure completamente opposte, e così via.

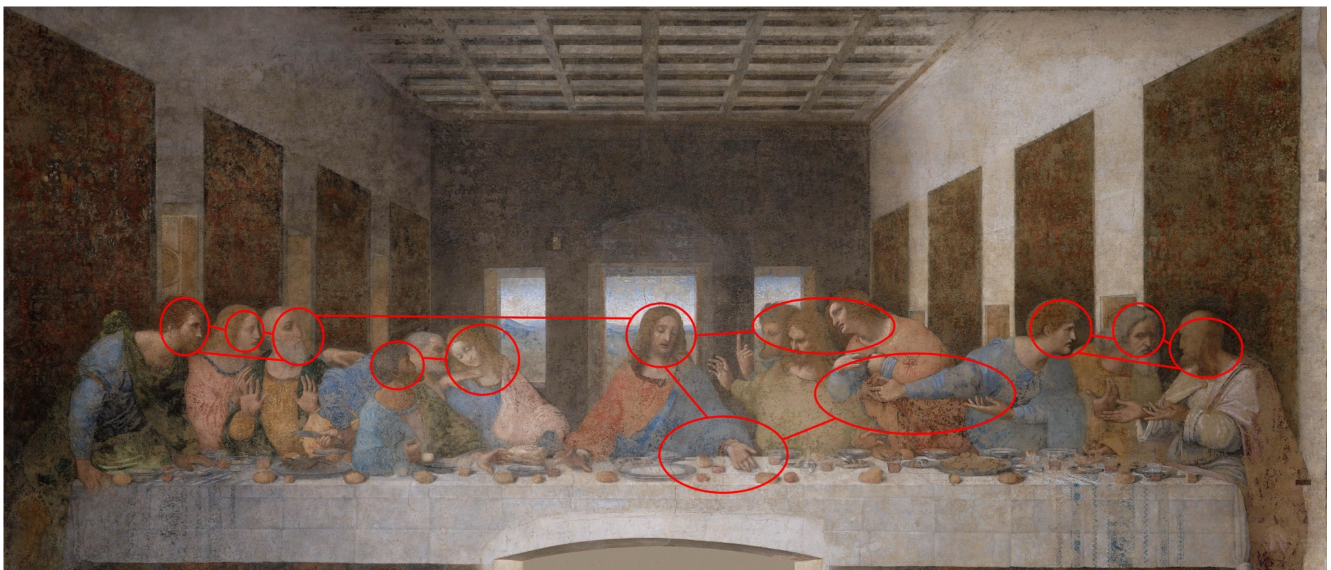


Figura 1: Cenacolo vinciano (annotato per evidenziare alcune delle zone di interesse e le loro relazioni).

Il report finale del progetto di ricerca sarà costituito da un elenco delle zone di interesse. Per ognuna verranno descritte, oltre alla zona in sè, anche tutte le relazioni in cui è coinvolta.

Come suo solito, l'amico di Luca è stato particolarmente prolisso e ora il professore a capo del progetto di ricerca non vuole accettare il suo lavoro finché non farà una drastica selezione, includendo al più **dieci** zone di interesse.

Dopo tutto questo lavoro di accurata descrizione delle relazioni, l'amico è costretto a obbedire al professore ma non vuole sacrificare nemmeno una delle descrizioni già prodotte: aiutalo a selezionare un sottoinsieme S delle N zone in modo che il report finale contenga comunque la descrizione di *tutte* le relazioni!

Implementazione

Dovrai sottoporre un unico file con estensione `.cpp` o `.c`.

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

Dovrai implementare la seguente funzione:

■ **Funzione `riassumi`**

C/C++	`int riassumi(int M, int M, int A[], int B[], int S[]);`

- L'intero N rappresenta il numero delle zone di interesse.
- L'intero M rappresenta il numero delle relazioni tra le zone.
- Gli array `A` e `B`, indicizzati da 0 a $M-1$, contengono alla posizione i la seguente informazione: esiste una relazione tra le zone `A[`i`]` e `B[`i`]`.
- La funzione dovrà restituire il numero S di zone da includere nel report finale e riempire l'array `S` (nelle posizioni da 0 a $S-1$) con gli identificativi di tali zone.

Il grader chiamerà la funzione `riassumi` in accordo a quanto specificato di seguito.

Allegata 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 da `stdin`, chiama la funzione che dovete implementare e scrive su `stdout`, secondo il seguente formato.

Il file di input è composto da $M+1$ righe, contenenti:

- Riga 1: gli interi N e M, separati da uno spazio.
- Righe $2, \ldots, M+1$: i due interi `A[`i`]` e `B[`i`]` per $i = 0, \ldots, M-1$.

Il file di output è composto da due righe, contenenti:

- Riga 1: il valore S restituito dalla funzione `riassumi`.
- Riga 2: i valori contenuti nell'array `S` (posizioni da 0 a $S-1$) così come modificato dalla funzione `riassumi`.

Assunzioni

- $2 \leq N \leq 50\,000$.
- $1 \leq M \leq 100\,000$.
- È sempre garantita l'esistenza di una soluzione, ovvero esiste sempre un sottoinsieme costituito da al più dieci zone che rispetta quanto richiesto.
- Se esistono più soluzioni è possibile stamparne una qualsiasi.
- L'ordine con cui vengono riportate le zone in una soluzione è irrilevante.
- Una relazione non viene mai ripetuta in input; inoltre $A_i \neq B_i$ per ogni $i = 0 \ldots M-1$.
- Ogni zona d'interesse è coinvolta in almeno una relazione.

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** [**0 punti**]: Casi d'esempio.
- **Subtask 2** [**40 punti**]: $N \leq 15$.
- **Subtask 3** [**40 punti**]: $N \leq 1000$.
- **Subtask 4** [**20 punti**]: Nessuna limitazione specifica.

Esempi di input/output

input	output
5 3 1 4 3 5 2 4	5 1 2 3 4 5
11 15 4 1 1 3 2 4 8 7 5 8 9 10 11 5 6 9 10 6 3 8 7 11 3 5 8 6 9 2 5 6	10 3 4 2 5 6 7 8 9 10 11

Spiegazione

Nel **primo caso di esempio** le zone di interesse sono così poche da poter essere incluse tutte nel report senza problemi. Naturalmente sarebbe stato ammissibile produrre soluzioni diverse (ad esempio includendo le sole zone 3 e 4).

Nel **secondo caso di esempio** ci sono $11 > 10$ zone di interesse, quindi dobbiamo escluderne (almeno) una. Possiamo escludere la zona 1, le cui uniche relazioni sono con le zone 3 e 4 già incluse nel report finale.

Soluzione

Questo problema è una versione mascherata del noto “problema di copertura dei vertici”, forse più conosciuto con la dicitura inglese *vertex-cover*.

Possiamo sintetizzare il problema come segue: dato un grafo $G = (V, E)$ si vuole trovare un sottoinsieme dei suoi vertici $S \subseteq V$ tale che ogni arco abbia almeno un estremo in S. In altre parole, per ogni $(u, v) \in E$ almeno un vertice tra u e v deve essere incluso nell'insieme S, che è detto *copertura*.

Questo problema è *NP-completo*, ovvero appartiene a una classe di problemi detta *NP*, per i quali possiamo verificare se una soluzione è ammissibile in tempo polinomiale. Nel caso di vertex cover, quindi, dato un insieme di vertici è “facile” verificare se è davvero una copertura. Essendo inoltre anche un problema *NP-hard*, si sospetta che per esso non esistano soluzioni polinomiali (a meno che *P=NP*)[3].

■ Soluzione esponenziale

Abbiamo accennato che per i problemi *NP* è “facile” verificare se la soluzione è corretta. Concentriamoci quindi ora sul nostro problema e descriviamo una semplice procedura che dato un insieme $S \subseteq V$, stabilisca se tale insieme è davvero una copertura valida per il grafo G. È sufficiente iterare su tutti gli elementi $s \in S$ e segnare come “visitati” tutti gli archi ad essi incidenti. Al termine, in tempo lineare nel numero dei vertici e degli archi, potremo affermare che S è una copertura valida se e solo se tutti gli archi sono stati visitati.

Non resta altro che provare “a forza bruta” tutte le possibili combinazioni con cui è possibile formare insiemi di al più dieci elementi partendo dagli N vertici del grafo. Uno dei modi più facili per implementare questa soluzione è far ricorso a una funzione ricorsiva che, richiamata via via su tutti i vertici, provi per ognuno ad includerlo oppure escluderlo dall'insieme candidato. Quando tale insieme raggiunge cardinalità dieci, si verifica se costituisce o meno una copertura tramite la procedura lineare descritta sopra.

Poiché le assunzioni garantiscono l'esistenza di tale copertura, esplorando tutte le possibilità siamo certi di trovarla, pagando purtroppo una complessità esponenziale.

■ Vertex-cover è trattabile fissato il parametro (FPT)

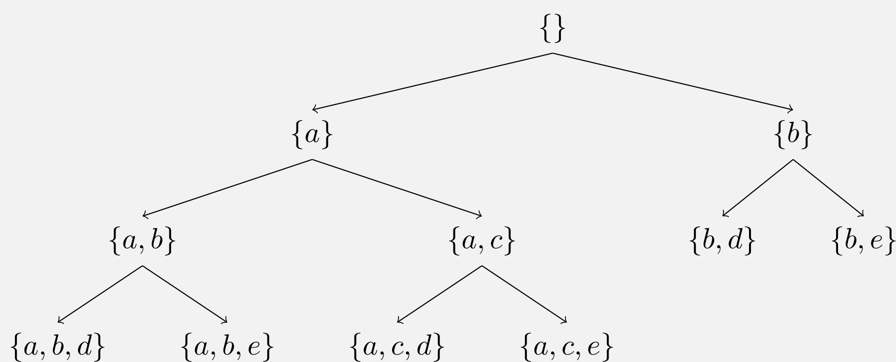
Con un po' di allenamento, ci si accorge che questo problema ha un'assunzione abbastanza singolare: qualunque sia la dimensione del grafo, esiste una copertura con al più dieci vertici. Questo da un lato può suggerire lo sviluppo di alcune euristiche greedy, sfruttando il fatto che i vertici nella copertura hanno probabilmente un alto numero di archi incidenti; dall'altro dovrebbe far riflettere sulla natura del problema.

Ribadiamo il fatto che siamo certi esista una copertura come richiesto. Preso allora un qualsiasi arco (a, b) sicuramente a deve appartenere alla copertura oppure b deve appartenervi. Non può essere che la copertura non contenga né a né b, perché altrimenti l'arco (a, b) non sarebbe coperto, contro la definizione di copertura. Possiamo quindi “esplorare” due possibili casi: la copertura contiene a oppure la copertura contiene b.

Proseguiamo allo stesso modo e consideriamo un arco (b, c): se prima la scelta è ricaduta sull'includere a, ora possiamo includere b oppure c arrivando ad avere $\{a, b\}$ oppure $\{a, c\}$. Se invece prima avessimo scelto di includere b, ora l'arco (b, c) sarebbe già coperto e non ci sarebbe bisogno di aggiungere nulla.

Consideriamo ancora un altro arco: (d, e). A prescindere dalle scelte fatte in precedenza, abbiamo di nuovo due possibilità per coprirlo: aggiungere d oppure e alla copertura. Visualizziamo tramite un albero la situazione descritta.

[3] Ovviamente questa trattazione delle classi di complessità computazionale teoriche è solo una brevissima sintesi, ulteriori approfondimenti possono essere trovati facilmente nei libri classici di algoritmica oppure su Wikipedia



Si può proseguire analizzando tutti gli archi e aggiungendo mano a mano le due possibilità (dove servono) per completare la copertura. Questa tecnica è detta *bounded search tree*: stiamo costruendo un albero di ricerca della copertura, ma esso è limitato dalla dimensione massima della copertura. La chiave per non far espandere all'infinito l'albero e rischiare di percorrere strade non convenienti è proprio data dal limite dei dieci vertici. Questo particolare albero infatti è limitato a profondità dieci; se una foglia a profondità dieci non contiene ancora un insieme che copre tutto il grafo, non ha senso proseguire.

D'altra parte è relativamente facile convincersi che se una copertura esiste, con questo metodo sicuramente viene trovata entro profondità dieci perché stiamo analizzando tutti i possibili insiemi sensati (non aggiungendo inutilmente entrambi gli estermi di un arco) fino a dimensione dieci.

Questa categoria di problemi è detta *FPT* (Fixed-Parameter Tractable): il problema generale è intrattabile, ma se imponiamo limiti sul "parametro" possiamo trovare una soluzione lineare nella dimensione dell'input ed esponenziale solo nel parametro. Nello specifico, per questo problema il parametro è la dimensione della copertura e la complessità della soluzione è $\mathcal{O}((N + M) \cdot 2^{10})$. Ciò non cancella i ragionamenti iniziali sull'intrattabilità del problema nella sua formulazione generale: stiamo semplicemente sfruttando un'assunzione importante sulla dimensione della soluzione del problema.

■ Un'ulteriore osservazione

Per raggiungere il punteggio pieno, è necessario adottare l'approccio con albero di ricerca limitato aggiungendo una piccola osservazione preliminare.

Ricordiamo che il *grado* di un vertice è dato dal numero di archi ad esso incidenti. Consideriamo un generico grafo in input e supponiamo di scoprire che alcuni suoi vertici hanno grado ≥ 11. Cosa possiamo dire rispetto a tali vertici?

L'osservazione è abbastanza facile. Supponiamo che $v \in V$ abbia grado 11. Se includiamo v nella copertura, abbiamo "sistemato" 11 archi. Se non includiamo v nella copertura, allora quegli archi non sono coperti e per *tutti* dovremo includere l'altro estremo nella copertura. Avremmo quindi che, solo per coprire quegli undici archi, dovremmo includere 11 vertici nella copertura e già solo questo sfora le assunzioni imposte.

Abbiamo quindi dimostrato che ogni vertice con grado almeno pari a 11 dev'essere obbligatoriamente incluso nella copertura di dimensione massima 10. Possiamo quindi analizzare in anticipo il grafo, aggiungere nell'insieme candidato a essere copertura i vertici con grado superiore a 10, rimuovere gli archi ad essi incidenti (perché già "sistemati") e procedere infine come descritto nella sezione precedente.

Esempio di codice C++11

```
#include <vector>
#include <set>

using namespace std;

const int MAXN = 50000;
const int MAXM = 100000;
const int MAXK = 10;

int N, M;
pair<int, int> archi[MAXM];
bool salta_arco[MAXM];
vector<int> grafo[MAXN];
bool soluzione_trovata;
set<int> soluzione;

bool verifica_copertura(set <int> copertura) {
    bool coperto[MAXM] = {};
    for (int v : copertura)
        for (int i = 0; i < M; i++)
            if (archi[i].first == v or archi[i].second == v)
                coperto[i] = true;
    for (int i = 0; i < M; i++)
        if (!coperto[i])
            return false;
    return true;
}

void esplora(int arco, set <int> copertura) {
    if (salta_arco[arco])
        esplora(arco + 1, copertura);
    else {
        if (!soluzione_trovata) {
            if (copertura.size() == MAXK or arco == M) {
                if (verifica_copertura(copertura)) {
                    soluzione_trovata = true;
                    soluzione = copertura;
                }
            }
            else {
                bool a_trovato = copertura.find(archi[arco].first) != copertura.end();
                bool b_trovato = copertura.find(archi[arco].second) != copertura.end();
                if (a_trovato or b_trovato)
                    esplora(arco + 1, copertura);
                else {
                    set<int> nuova_copertura_a = copertura;
                    set<int> nuova_copertura_b = copertura;
                    nuova_copertura_a.insert(archi[arco].first);
                    nuova_copertura_b.insert(archi[arco].second);
                    esplora(arco + 1, nuova_copertura_a);
                    esplora(arco + 1, nuova_copertura_b);
                }
            }
        }
    }
}

int riassumi(int N, int M, int A[], int B[], int S[]) {
    ::N = N; ::M = M;
    for (int i = 0; i < M; i++) {
        archi[i] = {A[i] - 1, B[i] - 1};
        grafo[A[i]-1].push_back(B[i]-1);
        grafo[B[i]-1].push_back(A[i]-1);
    }
    // Preprocessing: aggiungo alla copertura i vertici con grado >= 11.
    set<int> copertura;
    for (int i = 0; i < N; i++)
        if (grafo[i].size() >= 11)
            copertura.insert(i);
    for (int i = 0; i < M; i++)
        if (copertura.find(archi[i].first) != copertura.end() or
                copertura.find(archi[i].second) != copertura.end())
            salta_arco[i] = true;
    // Bounded search tree (FPT).
    esplora(0, copertura);
    // Restituisco la copertura trovata come soluzione.
    int i = 0;
    for (int v : soluzione)
        S[i++] = v + 1;
    return soluzione.size();
}
```