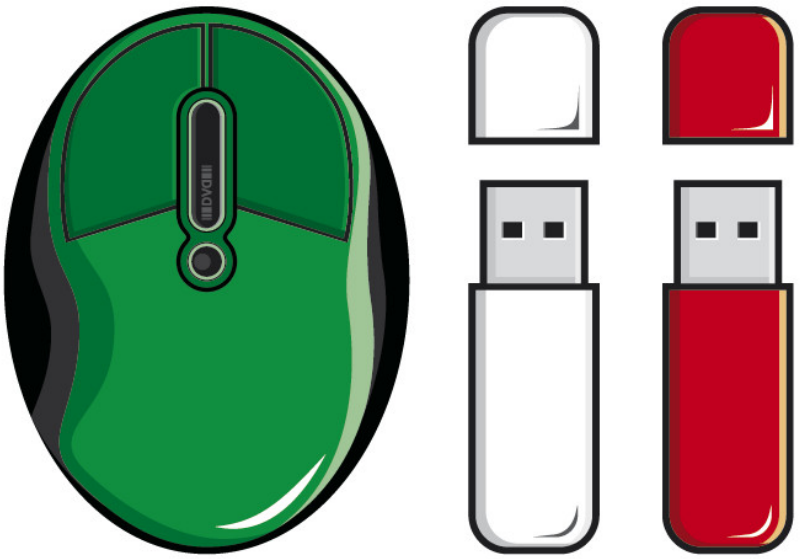
Ministero dell'Istruzione
dell'Università e Ricerca

AICA
Associazione Italiana per l'Informatica
ed il Calcolo Automatico



OLIMPIADI ITALIANE DI INFORMATICA

SALERNO

DUEMILA13

Risultati, Testi e Soluzioni ufficiali dei problemi

Problemi a cura di
Luigi Laura

Coordinamento
Monica Gati

Testi dei problemi
Giorgio Audrito, Matteo Boscariol, Roberto Grossi, Luigi Laura,
Giuseppe Ottaviano, Giovanni Paolini, Romeo Rizzi

Soluzioni dei problemi
William Di Luigi, Giovanni Paolini, Luca Versari

Supervisione a cura del Comitato per le Olimpiadi di Informatica

Indice

1 Risultati

Dal 12 al 14 settembre 2013, presso il campus universitario di Fisciano, si è svolta l'undicesima edizione delle finali delle Olimpiadi Italiane di Informatica: OII 2013.

Alla competizione, che richiedeva di individuare gli algoritmi per risolvere tre problemi complessi e tradurli utilizzando uno dei tre linguaggi di programmazione C, C++ e Pascal, hanno preso parte 87 atleti provenienti da tutta Italia, come riassunto in Tabella 1, che mostra come quest'anno la Lombardia abbia portato alla finale un numero davvero considerevole di atleti.

Regione	**N° atleti**
Lombardia	23
Emilia Romagna	11
Puglia	9
Veneto	9
Friuli Venezia Giulia	4
Lazio	4
Piemonte	4
Trentino Alto Adige	4
Sicilia	3
Toscana	3
Campania	2
Liguria	2
Marche	2
Molise	2
Abruzzo	1
Basilicata	1
Calabria	1
Sardegna	1
Umbria	1

Tabella 1: Atleti partecipanti alle OII 2013 per Regione di provenienza

La tabella 2 mostra invece la distribuzione degli atleti per tipologia di scuola superiore di provenienza. Come si può vedere, più della metà dei finalisti era iscritta a Istituti Tecnici Industriali, mentre Licei Scientifici e Istituti di Istruzione Superiore praticamente si dividevano il resto dei partecipanti.

Tipologia di Istituto	**N° atleti**
ITI	45
LS	19
IIS	18
ITC	5

Tabella 2: Atleti partecipanti alle OII 2013 per Tipologia di Istituto di provenienza

La distribuzione degli atleti per anno di nascita è invece riportata in Tabella 3.

Anno di nascita	**N° atleti**
1994	2
1995	49
1996	30
1997	6

Tabella 3: Atleti partecipanti alle OII 2013 per Anno di nascita

L'applicazione della regola di assegnazione delle medaglie (5 ori, 10 argenti e 20 bronzi) corretta per tenere conto degli ex-aequo ha portato ad assegnare 5 ori, 14 argenti e 17 bronzi, come riassunto in Tabella 4.

Regione	**Oro**	**Argento**	**Bronzo**	**Totale**
Lombardia		5	4	9
Emilia Romagna	1		3	4
Lazio	1	1	2	4
Puglia	1	1	1	3
Trentino Alto Adige		1	2	3
Veneto		2	1	3
Friuli Venezia Giulia		2		2
Piemonte			2	2
Calabria			1	1
Campania	1			1
Liguria		1		1
Molise	1			1
Sardegna			1	1
Toscana		1		1
Totale	5	14	17	36

Tabella 4: Distribuzione delle medaglie assegnate per Regione

Va sottolineato come il primo assoluto in classifica – Francesco Milizia dell'ITI Majorana di Brindisi – abbia totalizzato 292 dei 300 punti disponibili, e solo per un soffio (qualche frazione di secondo di troppo nel tempo di esecuzione dei suoi programmi) non abbia fatto l'en-plein.

Vanno anche ricordati Stefano Rando del Liceo Scientifico Gregorio Da Catino di Poggio Mirteto e Carlo Buccisano del Liceo Scientifico Lussana di Bergamo, giovanissimi medagliati di bronzo (nati il 30 e il 29 dicembre 1997) e Daniela Piras dell'ITI Othoca di Oristano, unica ragazza partecipante che ha pure ottenuto una medaglia di bronzo.

La competizione ha avuto uno svolgimento assolutamente regolare, in un ambiente di gara allestito in modo perfetto e con un'organizzazione a supporto degli atleti e dei loro accompagnatori davvero encomiabile.

Merito innanzitutto dell'Università degli Studi di Salerno-Fisciano e del Dipartimento di Studi e Ricerche Aziendali (Management & Information Technology) ma soprattutto della professoressa Genny Tortora e della

dottoressa Paola De Roberto, infaticabili coordinatrici di un magnifico staff di supporto alle OII. A tutti loro un sincero GRAZIE DI CUORE da tutto il Comitato Olimpico.

Nei ringraziamenti non possiamo naturalmente dimenticare gli enti locali che hanno collaborato all'organizzazione (il Comune di Baronissi, l'Istituto di Istruzione Superiore di Baronissi e il suo referente, il prof. Nicola Ansanelli, l'A.DI.S.U. – Azienda per il diritto allo studio universitario) e gli sponsor (l'USR Campania e le ditte Business Solution srl, Indra e Metoda Spa).

L'ospitalità e l'organizzazione sono state così eccellenti da convincere il Comitato Olimpico a fare tesoro degli sforzi e degli investimenti fatti, decidendo di tenere le OII 2014 ancora nella prestigiosa sede di Fisciano.

Prof. Nello Scarabottolo
Presidente Comitato Olimpiadi Italiane di Informatica

2 Mozzarelle di bufala (`bufale`)

2.1 Descrizione del problema

Salerno è la patria delle mozzarelle di bufala. A Monica e Paola sono state regalate N mozzarelle di bufala, tutte di tipologie diverse. Monica e Paola, dopo aver trascorso vari giorni a Salerno, hanno provato tutti i tipi di mozzarelle e hanno entrambe ormai sviluppato delle preferenze. Il vostro compito è quello di aiutare Monica e Paola a dividersi le mozzarelle.

In particolare, sia Monica che Paola hanno dato un voto, espresso da un numero intero maggiore o uguale a 0, ad ogni mozzarella. I voti, ovviamente, non sono necessariamente correlati. Dovete dividere le N mozzarelle (N è un intero positivo pari) in due gruppi di $\frac{N}{2}$ mozzarelle, uno per Monica e l'altro per Paola, in maniera che la somma complessiva dei voti (cioè, la somma dei voti di Monica relativa al gruppo di mozzarelle per Monica sommata alla somma dei voti di Paola relativa al gruppo di mozzarelle per Paola) sia massima.

Per esempio, supponiamo che ci siano le seguenti 8 mozzarelle, con i seguenti voti:

Numero mozzarella	1	2	3	4	5	6	7	8
Voto di Monica	10	2	4	6	1	7	3	4
Voto di Paola	6	6	1	0	3	8	5	7

In questo caso la soluzione migliore consiste nel dividere le mozzarelle nei due gruppi $(1, 3, 4, 6)$ per Monica, e $(2, 5, 7, 8)$ per Paola. In questo modo la somma totale è 48, ottenuta sommando 27 (dalle mozzarelle di Monica) con 21 (dalle mozzarelle di Paola). Non è possibile trovare una divisione delle mozzarelle che totalizzi un valore migliore.

Per risolvere il problema, dovete scrivere una funzione `solve`(N, M, P), che restituisca un singolo numero intero: il massimo valore ottenibile suddividendo tra Monica e Paola le N mozzarelle, sapendo che la i-esima mozzarella è valutata M_i da Monica e P_i da Paola.

2.2 Subtask

- **Subtask 0 [5 punti]:** caso di esempio.
- **Subtask 1 [7 punti]:** $2 \leqslant N \leqslant 5\,000$ e tutti i voti sono 0 oppure 1.
- **Subtask 2 [12 punti]:** $2 \leqslant N \leqslant 1\,000\,000$ e tutti i voti sono 0 oppure 1.
- **Subtask 3 [33 punti]:** $2 \leqslant N \leqslant 5\,000$ e tutti i voti sono compresi tra 0 e 10.
- **Subtask 4 [35 punti]:** $2 \leqslant N \leqslant 1\,000\,000$ e tutti i voti sono compresi tra 0 e 10.
- **Subtask 5 [8 punti]:** $2 \leqslant N \leqslant 10\,000\,000$ e tutti i voti sono compresi tra 0 e $10\,000\,000$.

2.3 Dettagli di implementazione

Dovrai sottoporre esattamente un file con estensione: `.c`, `.cpp` o `.pas`. Questo file deve implementare la funzione `solve` utilizzando uno dei seguenti prototipi.

Programma in C o C++

```
long long solve(int N, int* M, int* P);
```

Programma in Pascal

```
function solve(N: longint; var M: array of longint; var P: array of longint): int64;
```

La funzione implementata dovrà comportarsi come sopra descritto. Naturalmente sei libero di implementare altre routine ausiliarie per uso interno. La tua sottoposizione non deve effettuare direttamente operazioni di input o output, via console o via file di testo.

Funzionamento del grader di esempio

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 risultato restituito dalla vostra funzione sul file `output.txt`. Il file `input.txt` deve essere composto da $N + 1$ righe, in questo formato:

- nella prima riga deve essere presente un singolo intero, N;
- nella i-esima delle successive N righe devono essere presenti 2 interi: M_i e P_i.

2.4 Esempio di input/output

File `input.txt`	**File** `output.txt`
8 10 6 2 6 4 1 6 0 1 3 7 8 3 5 4 7	48

2.5 Descrizione della soluzione

- Se per una mozzarella i voti di Monica e Paola sono rispettivamente M_1 e P_1, e per un'altra sono M_2 e P_2, allora assegnando la prima mozzarella a Monica e la seconda a Paola si ottiene una felicità totale di $M_1 + P_2$, mentre facendo il contrario si ottiene una felicità totale di $M_2 + P_1$. Osserviamo che

$$M_1 + P_2 \geqslant M_2 + P_1 \quad \Longleftrightarrow \quad M_1 - P_1 \geqslant M_2 - P_2.$$

- Conviene quindi dare a Monica le mozzarelle che hanno differenza di voti più alta. Quindi basta ordinare le mozzarelle per $M_i - P_i$ e dare le prime $\frac{N}{2}$ a Monica e le altre $\frac{N}{2}$ a Paola.
- Per risolvere l'ultimo Subtask (soluzione lineare), si trova la $\frac{N}{2}$-esima mozzarella usando il "quick select" (esempio: con `nth_element` della STL), e si assegnano le mozzarelle di conseguenza.

2.6 Codice della soluzione (C++)

```cpp
#include <algorithm>
#define MAXN 10000000
using namespace std;

int D[MAXN];

long long solve(int N, int M[], int P[]) {
    long long somma = 0;
    for (int i=0; i<N; i++) {
        D[i] = M[i] - P[i];
        somma += M[i] + P[i];
    }
    nth_element(D, D+N/2, D+N);
    for (int i=0; i<N/2; i++)
        somma -= D[i];
    for (int i=N/2; i<N; i++)
        somma += D[i];
    return somma / 2;
}
```

2.7 Codice della soluzione (Pascal)

```pascal
unit bufale;

interface

function solve(N: longint; var M: array of longint; var P: array of longint): int64;

implementation

const MAXN = 10000000;
var D : array[0..MAXN] of longint;

(* implementazione dell'algoritmo "quickselect" *)
procedure nth_element(l, k, r: longint);
var
    i, j, x, tmp: longint;
begin
    i := l; j := r;
    x := D[l + random(r-l+1)];
    repeat
        while D[i] < x do inc(i);
        while x < D[j] do dec(j);
        if i <= j then
    begin
        tmp := D[i]; D[i] := D[j]; D[j] := tmp;
        inc(i); dec(j);
    end;
    until i > j;
    if j >= k then nth_element(l, k, j);
    if i <= k then nth_element(i, k, r);
end;

```

```
32  function solve(N: longint; var M: array of longint; var P: array of longint): int64;
33  var somma : int64;
34      i: longint;
35  begin
36      randseed := 123; (* seed per generare numeri pseudocasuali *)
37      somma := 0;
38      for i:=0 to N-1 do
39      begin
40          D[i] := M[i] - P[i];
41          inc(somma, M[i] + P[i]);
42      end;
43      nth_element(0, N div 2-1, N-1);
44      for i:=0 to (N div 2)-1 do
45          dec(somma, D[i]);
46      for i:=(N div 2) to N-1 do
47          inc(somma, D[i]);
48      solve := somma div 2;
49  end;
50
51  end.
```

3 Spiedini di frutta (`spiedini`)

3.1 Descrizione del problema

Ad Elisa piace molto la frutta, ma è anche molto esigente: il suo frutto preferito è la fragola, di cui non si stancherebbe mai; al contrario, riesce a mangiare solo un certo numero di altri frutti.

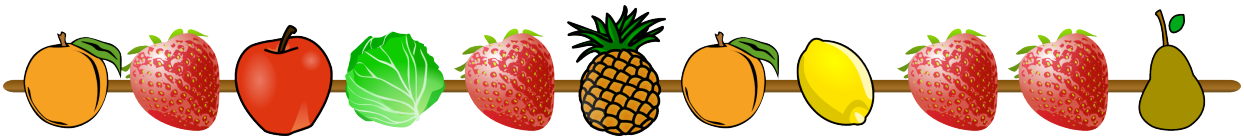
In particolare, a ogni frutto Elisa assegna un valore che indica quanto quel frutto non le piace: 0 per le fragole, altrimenti un numero intero compreso tra 1 (sopportabile) e 9 (lo detesta). In un pasto, Elisa può mangiare dei frutti solo se la somma dei loro valori non supera K.

A una cena, Elisa ordina uno spiedino di N frutti; sapendo che è possibile estrarre i frutti solo dagli estremi, e che Elisa deve mangiare ogni frutto che estrae dallo spiedino, aiutate Elisa a scegliere la strategia migliore per mangiare quante più fragole possibile.

Per esempio, consideriamo il seguente spiedino con 11 frutti, dove la prima riga indica le posizioni dei frutti nello spiedino, numerate da 0 a 10, mentre la seconda riga contiene il valore del frutto (ricorda che il valore 0 corrisponde sempre a una fragola):

0	1	2	3	4	5	6	7	8	9	10
1	0	2	8	0	5	1	6	0	0	3

Questa tabella descrive, per esempio, il seguente spiedino:



Supponiamo che K sia uguale a 9. La strategia migliore per Elisa è quella di mangiare i frutti fino alla posizione 1 da sinistra, e fino alla posizione 8 da destra. In questa maniera mangia 3 fragole (nelle posizioni 1, 8 e 9), mentre la somma dei valori dei frutti che non le piacciono è 4 (1 in posizione 0 e 3 in posizione 10). Per riuscire a mangiare anche l'ultima fragola sarebbe necessario $K \geqslant 14$.

Per risolvere il problema dovete scrivere una funzione `solve`(N, K, S) che restituisca un singolo numero intero: il massimo numero di fragole che Elisa può mangiare, sapendo che lo spiedino è composto da N frutti aventi valori $S_0, S_1, \ldots, S_{N-1}$, e sapendo che Elisa tollera al massimo un valore di K.

3.2 Assunzioni

Per tutti i subtask vale l'assunzione $0 \leqslant K \leqslant 1\,000\,000\,000$.

3.3 Subtask

- **Subtask 0 [5 punti]:** caso di esempio.
- **Subtask 1 [10 punti]:** $1 \leqslant N \leqslant 500$.
- **Subtask 2 [27 punti]:** $1 \leqslant N \leqslant 5\,000$.
- **Subtask 3 [48 punti]:** $1 \leqslant N \leqslant 1\,000\,000$.
- **Subtask 4 [10 punti]:** $1 \leqslant N \leqslant 20\,000\,000$.

3.4 Dettagli di implementazione

Dovrai sottoporre esattamente un file con estensione: `.c`, `.cpp` o `.pas`. Questo file deve implementare la funzione `solve` utilizzando uno dei seguenti prototipi.

Programma in C o C++

```
int solve(int N, int K, int* S);
```

Programma in Pascal

```
function solve(N: longint; K: longint; var S: array of longint): longint;
```

La funzione implementata dovrà comportarsi come sopra descritto. Naturalmente sei libero di implementare altre routine ausiliarie per uso interno. La tua sottoposizione non deve effettuare direttamente operazioni di input o output, via console o via file di testo.

Funzionamento del grader di esempio

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 risultato restituito dalla vostra funzione sul file `output.txt`. Il file `input.txt` deve essere composto da 2 righe, in questo formato:

- nella prima riga devono essere presenti 2 interi: N e K;
- nella seconda riga devono essere presenti N interi: i valori dei frutti: S_0, S_1, ..., S_{N-1}.

3.5 Esempio di input/output

File `input.txt`	**File** `output.txt`
11 9 1 0 2 8 0 5 1 6 0 0 3	3

3.6 Descrizione della soluzione

- Dato che si può mangiare frutta solo da destra e da sinistra, la soluzione si otterrà mangiando i k frutti a sinistra e gli h frutti a destra (per un'opportuna scelta di k e h).
- Una semplice soluzione $O(N^3)$ prova a sommare, per ogni coppia (k, h), tutti i valori dei k frutti a sinistra e tutti i valori degli h frutti a destra. Ogni qualvolta otteniamo una somma che Elisa riesce a sopportare, contiamo quante fragole abbiamo mangiato: il massimo che troviamo sarà la soluzione.
- Precalcolando le somme prefisse e le somme suffisse (ovvero le somme cumulative da sinistra e quelle da destra) si può ottimizzare la precedente soluzione ottenendo complessità $O(N^2)$.

- Possiamo velocizzare drasticamente la nostra soluzione notando che, per una scelta qualsiasi di k, possiamo eseguire una ricerca binaria sulla parte rimanente dell'array per trovare il valore più grande di h che produce una somma sopportabile anche dopo essere stata sommata a quella già prodotta dai primi k frutti. La complessità è quindi $O(N \log N)$.
- Per ridurre il numero di operazioni a $O(N)$ osserviamo che, facendo crescere k, h non può che diminuire (se mangio più frutti cattivi a sinistra, ne posso mangiare meno a destra). Quindi si può iterare su k (da 0 a N) in questo modo:
  1. Aumento k di 1. Di conseguenza mangio un frutto (potenzialmente cattivo) in più;
  2. Se ora la somma delle cattiverie supera quella consentita, diminuisco h fino a quando la somma torna ad essere nei limiti.
  3. Se $k < N$, torna al passo 1.

Il valore di k aumenta al massimo N volte, il valore di h diminuisce al massimo N volte: di conseguenza il numero di operazioni è lineare in N.

3.7 Codice della soluzione (C++)

```cpp
int zeriDestra, totDestra, zeriSinistra, totSinistra;

int solve(int N, int K, int s[]) {
    // inizialmente, supponiamo di aver mangiato tutto da destra
    for(int i=0; i<N; i++) {
        if (s[i] == 0)
            zeriDestra++;
        totDestra += s[i];
    }
    int p1 = 0; // puntatore sinistro
    int p2 = 0; // puntatore destro
    int risposta = 0;
    while (p1<N) {
        // finche' ho mangiato troppo, mangio un frutto in meno da destra
        while (p2<N && totSinistra+totDestra>K) {
            if(s[p2] == 0)
                zeriDestra--;
            totDestra -= s[p2++];
        }
        // se a destra mangio 0 frutti ma mangio comunque troppo, allora esci
        if (p2==N && totSinistra+totDestra>K)
            break;
        // ho trovato una coppia di indici candidata per essere quella ottimale
        if (p1<=p2 && zeriDestra+zeriSinistra>risposta)
            risposta = zeriDestra+zeriSinistra;
        // mangio un frutto in piu' da sinistra alla prossima iterazione
        if (s[p1] == 0)
            zeriSinistra++;
        totSinistra += s[p1++];
    }
    return risposta;
}
```

3.8 Codice della soluzione (Pascal)

```
unit spiedini;

interface

function solve(N: longint; K: longint; var S: array of longint): longint;

implementation

var zeriDestra, zeriSinistra, totDestra, totSinistra: longint;

function solve(N: longint; K: longint; var S: array of longint): longint;
var i, p1, p2, risposta : longint;
begin
    (* inizialmente, supponiamo di aver mangiato tutto da destra *)
    for i:=0 to N-1 do
    begin
        if S[i] = 0 then
            inc(zeriDestra);
        inc(totDestra, S[i]);
    end;
    p1 := 0; (* puntatore sinistro *)
    p2 := 0; (* puntatore destro *)
    risposta := 0;
    while p1 < N do
    begin
        (* finche' ho mangiato troppo, mangio un frutto in meno da destra *)
        while (p2 < N) and (totDestra+totSinistra>K) do
        begin
            if S[p2] = 0 then
                dec(zeriDestra);
            dec(totDestra, S[p2]);
            inc(p2);
        end;
        (* se a destra mangio 0 frutti ma mangio comunque troppo, allora esci *)
        if (p2 = N) and (totDestra+totSinistra>K) then
            break;
        (* ho trovato una coppia di indici candidata per essere quella ottimale *)
        if (p1 <= p2) and (zeriDestra+zeriSinistra > risposta) then
            risposta := zeriDestra+zeriSinistra;
        (* mangio un frutto in piu' da sinistra alla prossima iterazione *)
        if s[p1] = 0 then
            inc(zeriSinistra);
        inc(totSinistra, s[p1]);
        inc(p1);
    end;
    solve := risposta;
end;

end.
```

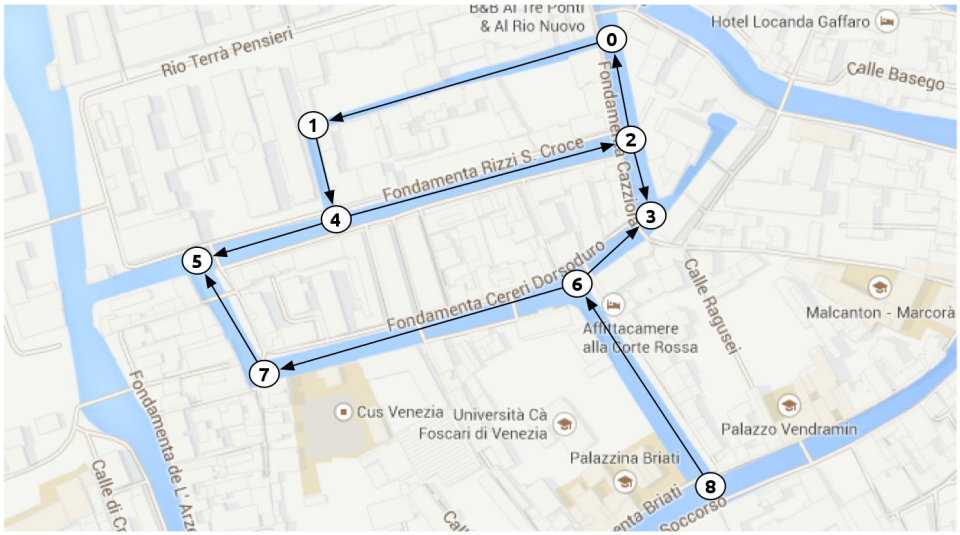
4 Maree a Venezia (`maree`)

4.1 Descrizione del problema

I gondolieri di Venezia si spostano lungo i canali, seguendo la corrente. Ogni canale è quindi percorribile in una sola direzione, nota ai gondolieri. Quando però arriva l'alta marea, le dinamiche della laguna sono tali che il verso della corrente di ogni canale si inverte. Dovete aiutare i gondolieri a pianificare il loro percorso per spostarsi in gondola da un punto di origine a un punto di destinazione, sapendo che sta per arrivare l'alta marea.

Data la mappa degli N punti di intersezione e degli M canali della città di Venezia che li collegano, con le correnti che inizialmente ne fluiscono, dovete determinare il minimo tempo necessario per spostarsi in gondola dal punto 0 al punto $N-1$, tenendo presente che:

- per attraversare un canale – cosa possibile solo nel senso della corrente – si impiega esattamente 1 minuto;
- dopo T minuti arriva l'alta marea e quindi la direzione di percorrenza di tutti i canali si inverte;
- nei punti di intersezione di due o più canali è possibile fermarsi per un numero di minuti a piacere.



Per esempio, considerate questa mappa. Per andare dal punto 0 al punto $N-1=8$, sapendo che l'alta marea arriverà al minuto $T=5$, si può seguire il seguente percorso (che richiede 7 minuti):

- da $t=0$ a $t=1$: $(0) \to (1)$
- da $t=1$ a $t=2$: $(1) \to (4)$
- da $t=2$ a $t=3$: $(4) \to (2)$
- da $t=3$ a $t=4$: $(2) \to (3)$
- da $t=4$ a $t=5$: si rimane fermi in (3)
- in $t=5$: alta marea
- da $t=5$ a $t=6$: $(3) \to (6)$
- da $t=6$ a $t=7$: $(6) \to (8)$

Un percorso alternativo, sebbene richieda 8 minuti e quindi non sia una soluzione valida, è il seguente:

- da $t = 0$ a $t = 1$: (0) → (1)
- da $t = 1$ a $t = 2$: (1) → (4)
- da $t = 2$ a $t = 3$: (4) → (5)
- da $t = 3$ a $t = 5$: si rimane fermi in (5)
- in $t = 5$: alta marea
- da $t = 5$ a $t = 6$: (5) → (7)
- da $t = 6$ a $t = 7$: (7) → (6)
- da $t = 7$ a $t = 8$: (6) → (8)

È facile verificare che, nella mappa mostrata, non ci sono modi di arrivare in $N - 1$ in meno di 7 minuti.

Per risolvere il problema, dovete scrivere una funzione `solve`(N, M, T, S, E), che restituisca un singolo numero intero: il minimo tempo necessario per arrivare dal punto 0 al punto $N - 1$, sapendo che l'i-esimo degli M canali collega i punti di intersezione S_i ed E_i, che la corrente inizialmente fluisce da S_i a E_i, e che l'alta marea arriva al minuto T. Se non è possibile andare dal punto 0 al punto $N - 1$, allora la funzione deve restituire il valore -1.

4.2 Subtask

- **Subtask 0 [5 punti]:** caso di esempio.
- **Subtask 1 [11 punti]:** $2 \leqslant N \leqslant 100\,000$, $1 \leqslant M \leqslant 2\,000\,000$ e $T = 0$.
- **Subtask 2 [25 punti]:** $2 \leqslant N \leqslant 3\,000$, $1 \leqslant M \leqslant 20\,000$ e $0 \leqslant T \leqslant 1\,000\,000\,000$.
- **Subtask 3 [38 punti]:** $2 \leqslant N \leqslant 100\,000$, $1 \leqslant M \leqslant 2\,000\,000$ e $0 \leqslant T \leqslant 1\,000\,000\,000$.
- **Subtask 4 [21 punti]:** $2 \leqslant N \leqslant 200\,000$, $1 \leqslant M \leqslant 3\,000\,000$ e $0 \leqslant T \leqslant 1\,000\,000\,000$.

4.3 Dettagli di implementazione

Dovrai sottoporre esattamente un file con estensione: `.c`, `.cpp` o `.pas`. Questo file deve implementare la funzione `solve` utilizzando uno dei seguenti prototipi.

Programma in C o C++

```
int solve(int N, int M, int T, int* S, int* E);
```

Programma in Pascal

```
function solve(N, M, T: longint; var S, E: array of longint): longint;
```

La funzione implementata dovrà comportarsi come sopra descritto. Naturalmente sei libero di implementare altre routine ausiliarie per uso interno. La tua sottoposizione non deve effettuare direttamente operazioni di input o output, via console o via file di testo.

Funzionamento del grader di esempio

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 risultato restituito dalla vostra funzione sul file `output.txt`. Il file `input.txt` deve essere composto da $M + 1$ righe, in questo formato:

- nella prima riga devono essere presenti 3 interi: N, M, T;
- nella i-esima delle successive M righe devono essere presenti 2 interi: S_i e E_i.

4.4 Esempio di input/output

File `input.txt`	**File** `output.txt`
9 10 5 0 1 1 4 4 2 2 0 2 3 4 5 7 5 6 7 6 3 8 6	7

Nota: l'esempio riportato è relativo alla situazione mostrata in figura.

4.5 Descrizione della soluzione

- Con una BFS a partire dall'incrocio 0 si calcola il tempo necessario per raggiungere ogni altro vertice (entro il tempo T di cambio di marea).
- Con un'altra BFS a partire dall'incrocio $N-1$ si calcola il tempo necessario per andare da ogni vertice al vertice $N-1$ con gli archi invertiti: per fare ciò con un'unica visita ci basta partire dal nodo $N-1$ ed usare il grafo normale (non trasposto).
- Ci sono due tipi di percorsi che giungono a destinazione: quelli che arrivano in fondo in tempo $\leqslant T$ e quelli che, invece, impiegano più tempo. Se esiste almeno un percorso che arriva in tempo $\leqslant T$, allora il problema è risolto (abbiamo già calcolato i percorsi minimi da 0 a ogni vertice che impiegano al più T minuti). Resta quindi da considerare il secondo caso.
- Chiamiamo v il vertice in cui ci si trova nel momento del cambio della marea. Il minimo tempo per giungere a destinazione è

  $$T + (\text{tempo per andare da } v \text{ a } N-1).$$

  Per concludere, basta scegliere il minimo tempo al variare di v.

Notiamo che il costo è dato da due BFS di costo $O(N + M)$ ciascuna e da un ciclo di costo $O(N)$ per trovare il nodo migliore sul quale vogliamo incontrare il cambio di marea. Il costo totale è quindi $O(N + M)$.

4.6 Codice della soluzione (C++)

```cpp
#include <vector>
#include <queue>
#include <limits>
#define MAXN 200000
using namespace std;

int N, M, T;

vector<int> graph[MAXN];
bool visited[MAXN];
int dist[MAXN];  // dist[x]  = distanza dal nodo 0 al nodo x
int rdist[MAXN]; // rdist[x] = distanza dal nodo N-1 al nodo x

// implementazione classica della visita in ampiezza
void bfs(int b, int* res) {
    queue< pair<int, int> > coda;
    coda.push(make_pair(b, 0));
    for (int i=0; i<N; i++)
        visited[i] = false;
    while (!coda.empty()) {
        pair<int, int> c = coda.front();
        coda.pop();
        if (visited[c.first])
            continue;
        visited[c.first] = true;
        res[c.first] = c.second;
        for (unsigned i=0; i<graph[c.first].size(); i++)
            coda.push(make_pair(graph[c.first][i], c.second+1));
    }
}

int solve(int n, int m, int t, int S[], int E[]) {
    N = n; M = m; T = t;
    for (int i=0; i<M; i++)
        graph[S[i]].push_back(E[i]);
    for (int i=0; i<N; i++)
        dist[i] = rdist[i] = numeric_limits<int>::max();
    bfs(0, dist);    // calcola le distanze partendo da 0
    bfs(N-1, rdist); // calcola le distanze partendo da N-1
    if (dist[N-1] <= T) // arriviamo a destinazione prima del cambio della marea?
        return dist[N-1];
    int best = numeric_limits<int>::max();
    // per ogni nodo, calcoliamo quanto costa arrivare li', attendere il
    // cambio della marea e successivamente arrivare al nodo N-1
    for (int i=0; i<N; i++)
        if (dist[i] <= T && rdist[i] < numeric_limits<int>::max()) {
            int v = T + rdist[i];
            if (v < best)
                best = v;
        }
    if (best == numeric_limits<int>::max()) // ho trovato almeno un percorso?
        return -1;
    return best;
}
```

4.7 Codice della soluzione (Pascal)

```
unit maree;

interface

function solve(N, M, T: longint; var S, E: array of longint): longint;

implementation

const MAXN  = 200000;
      QSIZE = 5000000;
var _n, _m, _t : longint;
    graph : array[0..MAXN-1] of array of longint;
    gsize, gcapa : array[0..MAXN-1] of longint;
    visited : array[0..MAXN-1] of boolean;
    dist, rdist : array[0..MAXN-1] of longint;
    q1, q2 : array[0..QSIZE-1] of longint;

(* implementazione classica della visita in ampiezza: per implementare
 * la coda in pascal facciamo uso di array circolari q1[] e q2[] in cui
 * "qhead" e' l'indice dell'elemento in testa alla coda, "qcount" e' il
 * numero di elementi presenti nella coda.
 *)
procedure bfs(b : longint; var res : array of longint);
var qhead, qcount, first, second, i, j : longint;
begin
    q1[0] := b; q2[0] := 0;
    qhead := 0; qcount := 1;
    for i:=0 to _n-1 do visited[i] := False;
    while qcount > 0 do
    begin
        first := q1[qhead];
        second := q2[qhead];
        inc(qhead);
        if qhead = QSIZE then
            qhead := 0;
        dec(qcount);
        if visited[first] then
            continue;
        visited[first] := True;
        res[first] := second;
        for j:=0 to gsize[first]-1 do
        begin
            q1[(qhead + qcount) mod QSIZE] := graph[first][j];
            q2[(qhead + qcount) mod QSIZE] := second + 1;
            inc(qcount);
        end;
    end;
end;

function solve(N, M, T: longint; var S, E: array of longint): longint;
var a, b, i, best : longint;
begin
    _n := N; _m := M; _t := T;
    for i:=0 to _n-1 do
```

```
55      begin
56          setlength(graph[i], 1);
57          (* all'inizio, la lista di adiacenza del nodo i ha dimensione 0
58           * e capacita' 1.
59           *)
60          gsize[i] := 0;
61          gcapa[i] := 1;
62          dist[i]  := maxlongint;
63          rdist[i] := maxlongint;
64      end;
65      for i:=0 to _m-1 do
66      begin
67          a := S[i]; b := E[i];
68          (* se ho esaurito i posti nella lista di adiacenza del nodo a *)
69          if gsize[a] = gcapa[a] then
70          begin
71              (* allora raddoppio la sua capacita' *)
72              gcapa[a] := gcapa[a] shl 1;
73              setlength(graph[a], gcapa[a]);
74          end;
75          graph[a][gsize[a]] := b;
76          inc(gsize[a]);
77      end;
78      bfs(0, dist);      (* calcola le distanze partendo da 0    *)
79      bfs(_n-1, rdist);  (* calcola le distanze partendo da N-1  *)
80      (* arriviamo a destinazione prima del cambio della marea?  *)
81      if dist[_n-1] <= _t then
82      begin
83          solve := dist[_n-1];
84          exit;
85      end;
86      best := maxlongint;
87      (* per ogni nodo, calcoliamo quanto costa arrivare li', attendere il
88       * cambio della marea e successivamente arrivare al nodo N-1
89       *)
90      for i:=0 to _n-1 do
91          if (dist[i] <= _t) and (rdist[i] < maxlongint) then
92              if _t + rdist[i] < best then
93                  best := _t + rdist[i];
94      (* ho trovato almeno un percorso? *)
95      if best = maxlongint then
96          solve := -1
97      else
98          solve := best;
99  end;
100
101 end.
```