

Unità didattica: Applicazione allo string matching

[2-AC]

Titolo: Ricerca di un pattern in un testo

Argomenti trattati:

- ✓ Ricerca della prima occorrenza di un pattern in un testo tutto in memoria
- ✓ Algoritmo di string matching con le funzioni C per manipolare le stringhe
- ✓ Algoritmo naive per la ricerca di un pattern in un testo tutto in memoria

Prerequisiti richiesti: algoritmo di ricerca sequenziale, fondamentali della programmazione C, tipo carattere e tipo stringa in C

String matching

Esempio: ricerca di un pattern in un testo

Si vuole cercare la prima occorrenza del carattere iniziale “i” del pattern e la prima occorrenza del pattern “ita” nel seguente testo

Nel mezzo
del cammin di nostra vita
mi ritrovai
per una selva oscura
che la diritta via
era smarrita

...

Dante Alighieri, Inferno

Versione 1a: con le funzioni C di manipolazione di stringhe

```
#include <stdio.h>
#include <string.h>
```

```
void main()
```

```
{char *patt = "ita", *ch; short i,j;
```

```
char *testo = "Nel mezzo del cammin di nostra vita\n"
              "mi ritrovai per una selva oscura\n"
              "che la diritta via era smarrita\n";
```

```
ch = strchr(testo,patt[0]);
```

trova prima occorrenza di "i"

```
printf("strchr(testo,patt[0])=%d\n",ch);
```

```
if (ch != 0) { j=ch-testo; printf("&testo[j]=%d\n",&testo[j]);
               printf(" *ch= "); putchar(*ch);
               printf(" coincide con *(testo+j)= "); putchar(*(testo+j));}
```

```
ch = strstr(testo,patt);
```

trova prima occorrenza di "ita"

```
printf("\nstrstr(testo,patt)=%d\n",ch);
```

```
if (ch != 0) { j=ch-testo; printf("&testo[j]=%d\n",&testo[j]);
               for (i=0; i < strlen(patt); i++) putchar(*(ch+i));
               /* ***** attenzione: non *ch+i !!! */
               printf(" coincide con ");
               for (i=j; i < j+strlen(patt); i++) putchar(*(testo+i));}
```

```
}
```

determina indice sull'array come differenza di indirizzi di memoria

"ita"

"Nel mezzo del cammin di nostra vita\n"
"mi ritrovai per una selva oscura\n"
"che la diritta via era smarrita\n"

output del programma precedente

```
strchr(testo,patt[0]) = 4198946  
&testo[j]=4198946  
*ch = i coincide con *(testo+j) = i  
strstr(testo,patt) = 4198960  
&testo[j]=4198960  
ita coincide con ita
```

*indirizzi
di
memoria*

*contenuto degli
indirizzi di memoria*

01234567890123456789012345678901234

```
*testo = "Nel mezzo del cammin di nostra vita\n"
"mi ritrovai per una selva oscura\n"
"che la diritta via era smarrita\n"
```

```
#include <stdio.h>
#include <string.h>
void main()
{
    char *patt = "ita", *p0 = "i", *ch; short i,j;
    char *testo = "Nel mezzo del cammin di nostra vita"
                  "mi ritrovai per una selva oscura"
                  "che la diritta via era smarrita";

    ch = strchr(testo, patt[0]);
    printf("\nstrchr(testo,patt[0]) = %d\n",ch);

    i = strcspn(testo, p0);
    printf("\nstrcspn(testo,p0) = %d\n",i);

    ...}

```

```
*patt="ita"
*p0="i"
```

strchr(testo,patt[0]) = 4198946

strcspn(testo,p0) = 18

Versione 1b

organizzazione dei dati: invece di ...

```
char *patt =
char *testo =
```

puntatore a stringa costante

si può usare ...

```
#include ...
```

```
void main()
```

```
{char patt[ ] = "ita", *ch; short i,j;
```

```
char testo[ ] = "Nel mezzo del cammin di nostra vita\n"
               "mi ritrovai per una selva oscura\n"
               "che la dritta via era smarrita\n";
```

array contenente i caratteri della stringa costante

```
...
```

	patt	0x0012ff74	"ita"
—	[0]	105	'i'
—	[1]	116	't'
—	[2]	97	'a'
—	[3]	0	''

String matching

Esempio: ricerca di un pattern in un testo

Si vogliono cercare tutte le occorrenze del pattern "ita" nel seguente testo

Nel mezzo
del cammin di nostra vita
mi ritrovai
per una selva oscura
che la diritta via
era smarrita

...

Dante Alighieri, Inferno

algoritmo **naive** di string matching

Nel mezzo del cammin di nostra
vita mi ritrovai per una selva oscura
che la diritta via era smarrita

In **testo** **pattern**
compare 2 volte

Idea: L'algoritmo più semplice (**naive**) di *string matching* consiste nel confrontare ogni carattere del **testo** con il primo carattere del **pattern** fino a quando il **testo** finisce. Ogni volta che questi due caratteri sono uguali si avvanza, sul testo e sul pattern, a confrontare i caratteri successivi.

Nel	mezzo	del	cammin	di	nostra	vita
ita	ritrovai	per	una	selva	oscura	
ita	la	diritta	via	era	smarrita	

versione 2:

intero testo in memoria e ricerca diretta

```
#include <stdio.h>
#include <string.h>
void main()
{char *testo ="Nel mezzo del cammin di nostra vita\n"
    "mi ritrovai per una selva oscura\n"
    "che la diritta via era smarrita.\n"
    "Ah quanto a dir qual era è cosa dura\n"
    "esta selva selvaggia e aspra e forte\n"
    "che nel pensier rinova la paura!\n";

    char *patt = "ita";
    int len_t,len_p;short it,itt,ip,trovato,num_trovati;
    printf("cerca pattern = %s\nin testo\n%s",patt,testo);
```

----- *...algoritmo...* -----

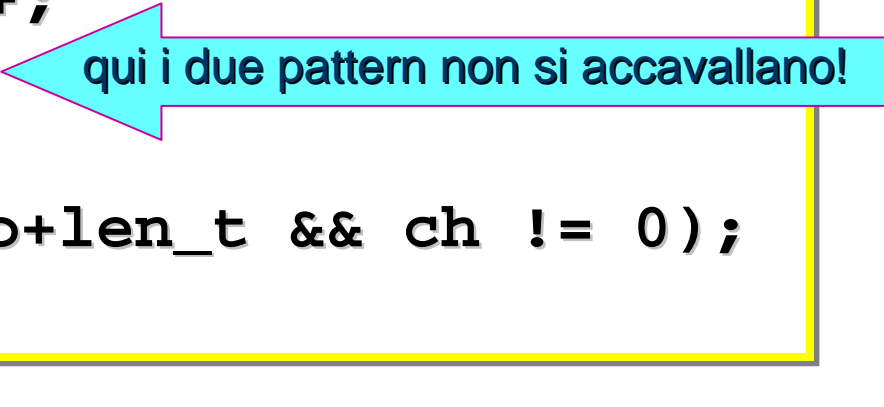
```
printf("pattern trovati = %d\n",num_trovati);
}
```

versione 2a (con funzioni C per gestire le stringhe)

```
...char *patt = "ita", *te,*ch;
int len_t,len_p;short num_trovati;

...
len_t=strlen(testo); len_p=strlen(patt);
num_trovati=0; te=testo;
do
{
    ch=strstr(te,patt);
    if (ch != 0)
    {
        num_trovati++;
        te=ch+len_p;
    }
} while (te < testo+len_t && ch != 0);

...
```



miglioramento: `te < testo + len_t - len_p + 1`

versione 2b (ricerca diretta)

04_02.11

```
... len_t=strlen(testo); len_p=strlen(patt);
it=ip=0; num_trovati=0; /* inizializzazione */
while (it < len_t) /* finché il testo non è finito */
{ if (testo[it] == patt[ip]) /* se il primo carattere è uguale */
  { uguali=1; itt=it+1; ip++; /* confronta anche i successivi */
    while (itt<len_t && ip<len_p && uguali)
    { if (testo[itt] == patt[ip]) uguali++;
      else uguali=0;
      itt++; ip++;
    }
    if (uguali == len_p)
      num_trovati=num_trovati+1;
  } /* trovato l'intero pattern */
  ip=0; it++; /* cerca il pattern dal carattere successivo */
}; ...
```

testo non finito

pattern non finito

... si può migliorare!

Complessità computazionale $T(M,N)=...$

(prof. M. R.)

Esempio: passi dell'algoritmo di ricerca diretta

testo →

a	b	b	a	a	a	b	a	b	b	a	b	a	b	b	b	a	b	a	b	b	a	a	a	b	b	b	a	a
a	b	a	b	b	a	a	a	← pattern																				
	a	b	a	b	b	a	a	a																				
		a	b	a	b	b	a	a	a																			
			a	b	a	b	b	a	a	a																		
				a	b	a	b	b	a	a	a																	
					a	b	a	b	b	a	a	a																
						a	b	a	b	b	a	a	a															
							a	b	a	b	b	a	a	a														
								a	b	a	b	b	a	a	a													
									a	b	a	b	b	a	a	a												
										a	b	a	b	b	a	a	a											
											a	b	a	b	b	a	a	a										
												a	b	a	b	b	a	a	a									
													a	b	a	b	b	a	a	a								
														a	b	a	b	b	a	a	a							
															a	b	a	b	b	a	a	a						
																a	b	a	b	b	a	a	a					
																	a	b	a	b	b	a	a	a				
																		a	b	a	b	b	a	a	a			

legenda

char	uguali
char	diversi

<i>legenda</i>	
char	uguali
char	diversi

...cont. passi dell'algoritmo di ricerca diretta

a	b	b	a	a	a	b	a	b	b	a	b	a	b	b	b	a	b	a	a	a	b	b	b	a	a
											a	b	a	b	b	a	a	a							
											a	b	a	b	b	a	a	a							
											a	b	a	b	b	a	a	a							
											a	b	a	b	b	a	a	a							
											a	b	a	b	b	a	a	a							
											a	b	a	b	b	a	a	a							
											a	b	a	b	b	a	a	a							
											a	b	a	b	b	a	a	a							

Numero confronti totali: 17 (passi) + 26 = 43

Algoritmo inefficiente!

complessità computazionale

L'algoritmo di ricerca diretta richiede, nel caso peggiore, NM confronti tra caratteri, dove N è la lunghezza del **testo** ed M quella del **pattern**.

In problemi di *text editing* (elaborazione di testi), dove $N \gg M$, l'algoritmo richiederà mediamente un numero di confronti $\approx N$, in quanto difficilmente il ciclo interno verrà eseguito.

Tuttavia quando l'**alfabeto** è **binario** oppure il **testo** è **altamente ripetitivo** la ricerca diretta risulta estremamente lenta: è conveniente pertanto ricorrere ad algoritmi più veloci.

Algoritmi di string matching



Naive (ricerca diretta);



Smart (automa a stata finiti);



Knuth-Morris-Pratt;



Boyer-Moore.



Esempio 3: cercare tutte le occorrenze del pattern
"ita" nel testo per eliminarle (versione "dinamica")

idea algoritmo

Nel mezzo del cammin di nostra
vita mi ritrovai per una selva oscura
che la diritta via era smarrita

Trova!

Nel mezzo del cammin di nostra
v mi ritrovai per una selva oscura
che la diritta via era smarr**ita**

Trova!

Nel mezzo del cammin di nostra
v mi ritrovai per una selva oscura
che la diritta via era smarr

Elimina!

idea algoritmo:

- 1) Costruisce stringa dinamica;
- 2) Trova ed elimina ogni occorrenza;

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
main()
```

```
{char *patt = "ita", *testo, *te, *ch;
int len_t, len_p; short num_trovati=0, num_byte;
```

```
testo=(char *)malloc(120);
```

```
strcpy(testo, "Nel mezzo del cammin di nostra vita");
```

```
printf("1) testo dopo strcpy(testo, ... =%s", testo);
```

```
strcat(testo, "mi ritrovai per una selva oscura");
```

```
printf("2) testo dopo strcat(testo, ... =%s", testo);
```

```
strcat(testo, "che la diritta via era smarrita");
```

```
printf("3) testo finale: ");
```

```
...
```

1) Costruisce stringa dinamica

```
1) testo dopo strcpy(testo, ... =
Nel mezzo del cammin di nostra vita
```

```
2) testo dopo strcat(testo, ... =
Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura
```

```
3) testo finale: Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura
che la diritta via era smarrita
```

2) Trova ed elimina ogni occorrenza

...

```
len_t=strlen(testo); len_p=strlen(patt); te=testo;
char *src, *dest;
do {ch=strstr(te,patt);
    if (ch != 0)
    { num_trovati++;
      src=ch+len_p; dest=ch;
      num_byte=len_t-(int)(ch-testo);
      memmove(dest, src, num_byte);
      te=ch; len_t=len_t-len_p;
    }
} while (te < testo+len_t-len_p+1 && ch != 0);
printf("pattern trovati = %d\n",num_trovati);
realloc(test, len_t);
}
```

Nel mezzo del cammin di nostra v
mi ritrovai per una selva oscura
che la diritta via era smarrita

Nel mezzo del cammin di nostra v
mi ritrovai per una selva oscura
che la diritta via era smarr

numero pattern trovati = 2

... si può migliorare!

evitando di spostare più volte
uno stesso blocco di memoria

... si può migliorare!

evitando di spostare più volte
uno stesso blocco di memoria

testo

Nel mezzo del cammin di nostra vita \nmi ritrovai per una selva oscura \nche la diritta via era
smarrita \n \n

Nel mezzo del cammin di nostra vita \nmi ritrovai per una selva oscura \nche la diritta via era s
marrita \n \n

Nel mezzo del cammin di nostra vita \nmi ritrovai per una selva oscura \nche la diritta via era s
marrita \n \n

Nel mezzo del cammin di nostra vita \nmi ritrovai per una selva oscura \nche la diritta via era s
mar \n \n

è stato spostato 2 volte!

... si può migliorare!

evitando di spostare più volte
uno stesso blocco di memoria

- 1) Trovare tutte le occorrenze
- 2) Rimuoverle e compattare

patt → **ura\0**

3 occorrenze in posizione: 10, 20, 31

testo

1^a occ

selva oscura cosa durata la paura rinova!\0

selva osc cosa durata la paura rinova!\0

2^a occ

selva osc cosa durata la paura rinova!\0

selva osc cosa dta la paura rinova!\0

3^a occ

selva osc cosa dta la paura rinova!\0

selva osc cosa dta la pa rinova!\0

rinova!\0

è stato spostato 3 volte!

ta la pa

è stato spostato 2 volte!

cosa d

è stato spostato 1 volte!

... miglioramento

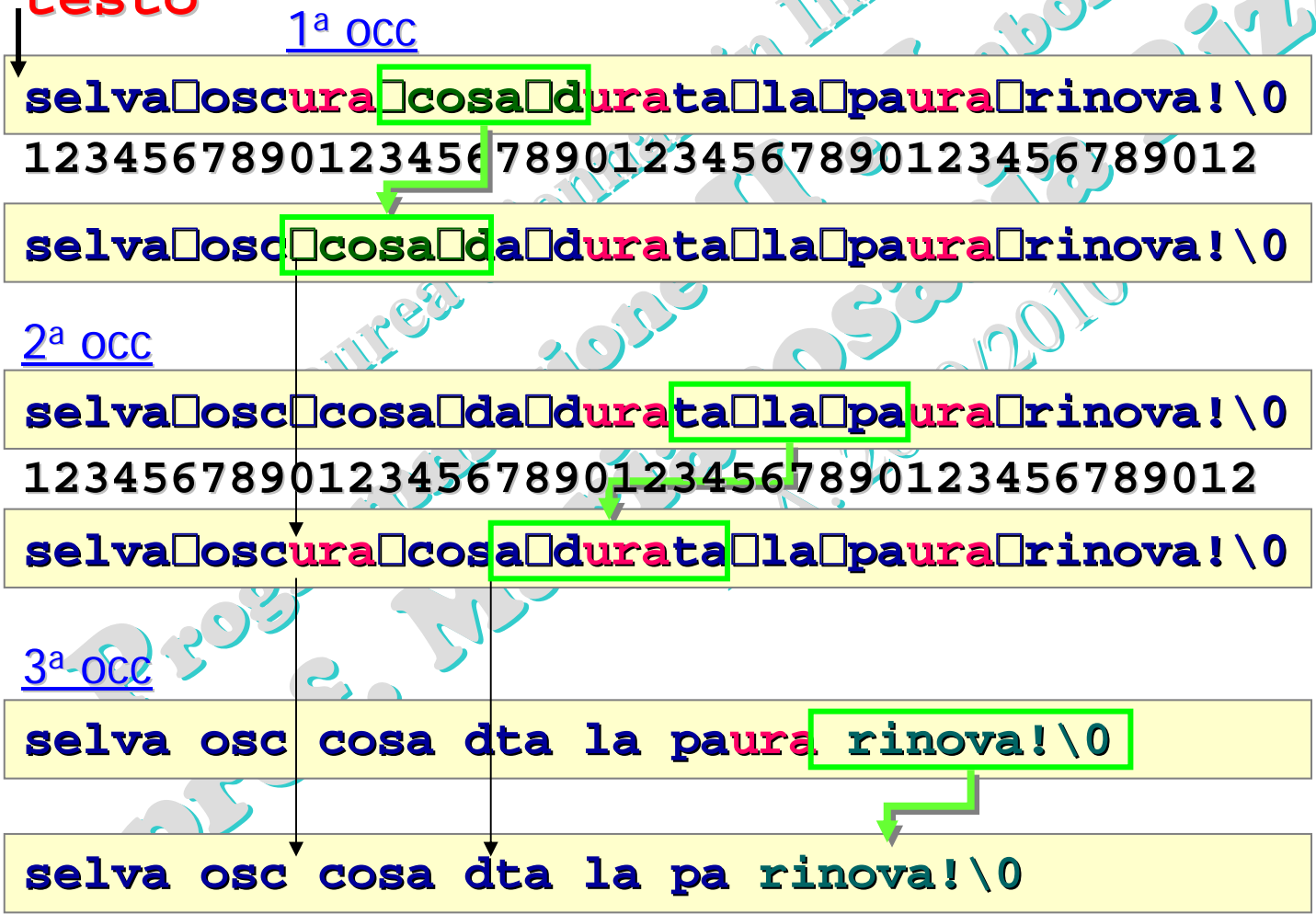
evita di spostare più volte uno stesso blocco di memoria

- 1) Trovare tutte le occorrenze
- 2) Rimuoverle e compattare

patt → **ura\0**

3 occorrenze in posizione: 10, 20, 31

testo



cosa d è stato spostato 1 volte!

ta la pa è stato spostato 2 volte!

rinova!\0 è stato spostato 3 volte!

Altre funzioni sulle stringhe

per manipolare byte (buffer manipulation)

(in `<string.h>`)

`void *dest, *src; char ch; intero num_byte`

<code>memcpy(dest, src, num_byte)</code>	copia il blocco di <code>num_byte</code> byte dalla posizione <code>src</code> alla posizione <code>dest</code>
<code>memmove(dest, src, num_byte)</code>	sposta il blocco di <code>num_byte</code> byte dalla posizione <code>src</code> alla posizione <code>dest</code>
<code>memset(dest, ch, num_byte)</code>	assegna al blocco di <code>num_byte</code> byte dalla posizione <code>dest</code> il valore del byte <code>ch</code>

Esercizi:...

1 Confrontando i risultati con le relative funzioni del C scrivere *function* C che restituisca la prima occorrenza di una sottostringa in una stringa senza usare `strstr(...)`.

2 Usando l'allocazione dinamica e le funzioni C per manipolare le stringhe, scrivere *function* C che restituisca la posizione di tutte le occorrenze di una sottostringa in una stringa ed il loro numero totale.

Esempio: cercare **"ita"** in **"vitalita`"**, conduce alle due occorrenze **"vitalita`"** con posizioni relative {2, 6}. **[liv. 1]**

Esercizi: Utilizzando per le stringhe

- l'allocazione statica
- l'allocazione dinamica

...

3

... scrivere *function C* che elimini tutte le occorrenze di una data sottostringa in una stringa col **minimo** numero di spostamenti di blocchi di memoria. [liv. 2]

4

... scrivere *function C* che sostituisca in un testo tutte le occorrenze di una data sottostringa S_1 con un'altra S_2 (le due sottostringhe possono avere anche lunghezze diverse). [liv. 2] ... col **minimo** numero di spostamenti di blocchi di memoria. [liv. 3]

Help!

Help!

facile!

Bisogna distinguere:

A. $\text{strlen}(S_1) == \text{strlen}(S_2)$

B. $\text{strlen}(S_1) > \text{strlen}(S_2)$

C. $\text{strlen}(S_1) < \text{strlen}(S_2)$

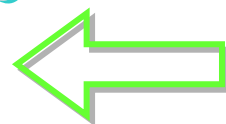
Negli altri due casi:

- 1) Trovare tutte le occorrenze
- 2) Sostituire le occorrenze

Inoltre nel caso...:

B): il testo si accorcia

C): il testo si allunga



... simile all'esercizio
di eliminazione

Se len_t indica la lunghezza iniziale del testo, num_occ il numero delle occorrenze e $d = \text{strlen}(S_2) - \text{strlen}(S_1)$, allora la lunghezza finale del testo è: $\text{len}_t + d * \text{num_occ}$

Nel caso...:

B): il testo si accorcia:

B.1) si modifica il testo dalla prima occorrenza
all'ultima;

B.2) si "rialloca" l'array accorciandolo.

C): il testo si allunga:

C.1) si "rialloca" l'array allungandolo;

C.2) si modifica il testo dall'ultima occorrenza
alla prima.

Se J_occ è l'array contenente le posizioni relative delle occorrenze di S_1 , conviene aggiungergli un'ultima componente contenente il valore "fittizio" len_t+1 (come se ci fosse un'altra occorrenza dopo la fine del testo!)

In tal modo l'algoritmo appare come un ciclo iterativo che agisce sulla porzione di testo compresa tra due occorrenze.

Esempio C): il testo si allunga

(con il minimo numero di spostamenti)

$s_1 \rightarrow$ ura

$s_2 \rightarrow$ XXXXX

3 occorrenze in posizione: Jocc

10
20
31
42

$\text{len}_t=41, \text{len}_1=3, \text{len}_2=5, d=2$

$\text{new_len}_t = 41+3*2=47$

testo

12345678901234567890123456789012345678901
selva_oscura_cosa_durata_la_paura_rinova!\0

rialloca

Jocc
10
20

3+31
42

sposta

selva_oscura_cosa_durata_la_paura_rinov_rinova!\0

inserisce S_2

Jocc
10

3+20
31
42

sposta

selva_oscura_cosa_durata_lta_la_paxxxx_rinova!\0

inserisce S_2

...

selva_oscura_cosa_dxxxx_lta_la_paxxxx_rinova!\0

Lunghezza testo = 41

1 2 3 4

12345678901234567890123456789012345678901234567

selva_oscura_cosa_durata_la_paura_rinova!

Occorrenze in posizione: 10 20 31

Lunghezza testo dopo sostituzione = 47

passo: 1

12345678901234567890123456789012345678901234567

selva_oscura_cosa_durata_la_paura_rinova_rinova!

12345678901234567890123456789012345678901234567

selva_oscura_cosa_durata_la_paura_XXXXXX_rinova!

passo: 2

12345678901234567890123456789012345678901234567

selva_oscura_cosa_durata_la_paxxxxx_rinova!

12345678901234567890123456789012345678901234567

selva_oscura_cosa_durata_la_paxxxxx_rinova!

passo: 3

12345678901234567890123456789012345678901234567

selva_oscura_cosa_dxxxxxta_la_paxxxxx_rinova!

12345678901234567890123456789012345678901234567

selva_oscura_cosa_dxxxxxta_la_paxxxxx_rinova!

$d = \text{strlen}(S_2) - \text{strlen}(S_1) = 2$

sposta: 34... → 40...

sposta: 23... → 27...

sposta: 13... → 15...

Jocc

10

20

31

42