Università di Perugia
Dipartimento di Matematica e Informatica

Relazione progetto Artificial Intelligent Systems:
Intelligent Application Development

Ricerca massima sottosequenza comune con ricerca in profondità

Studente
Cerami Cristian, 362384

Prof.
Marcugini Stefano

Anno Accademico 2022-2023

Indice

Capitolo 1

Introduzione

1.1 Obiettivo

L'obiettivo di questo progetto è la creazione di un codice, in linguaggio **OCaml**, che permetta di risolvere il problema intitolato "**Massima sottosequenza comune (ricerca in profondità)**" presente su Unistudim al seguente link: `https://unistudium.unipg.it/unistudium//mod/resource/view.php?id=389580` .

La richiesta di tale problema è la seguente:

> Si consideri un insieme finito S di stringhe ed un intero K.
> Determinare, se esiste, una stringa x di lunghezza maggiore o uguale a K che sia sottosequenza di ogni stringa s∈S .
> Si risolva il problema utilizzando una ricerca in profondità.

Nel capitolo successivo mostrerò come sono andato a risolvere il problema illustrando le funzioni create e spiegando a cosa servano.

1.2 Ambiente di lavoro

Il progetto è stato realizzato su **Windows** dove sono andato ad emulare l'ambiente **Unix** tramite il software **Cygwin**. Per installarlo, insieme ad OCaml, sono andato a sfruttare la procedura di installazione grafica disponibile al seguente link: `https://fdopen.github.io/opam-repository-mingw/installation/`

1.3 Descrizione del problema

Trovare una sottosequenza (o substring) in comune, date in input almeno due stringhe e una lunghezza minima K, vuol dire ricercare se all'interno di quelle stringhe vi sia una stessa sequenza ordinata di caratteri che abbia almeno lunghezza K. Di seguito possiamo vedere un esempio in cui vengono riportate **tutte** le substring in comune (`K>=1`):

```
["ciao"; "cioaiao"] = ["c";"i";"a";"o";"ci";"ia";"ao";"iao"]
```

Ovviamente esistono vari approcci per risolvere il problema; quello classico è il "*test and compare*" in cui, ad esempio, prima andiamo a generarci tutte le substring di ogni stringa in input e poi andiamo a verificare quali sono le substring, di lunghezza almeno K, in comune tra tutte quelle generate. Questa tecnica purtroppo non è efficace in quanto inizialmente andiamo a generare tutto l'insieme di substring in cui sono incluse anche quelle che sicuramente non ci permettono di arrivare alla soluzione. Un approccio migliore, che ci permette di risolvere questo problema, è il "***backtracking***". Con esso infatti si costruisce la sequenza soluzione aggiungendo un elemento alla volta e si usa un criterio per controllare se la sequenza parziale ha possibilità di successo:

- Ad ogni stadio `i`: se `(x1, ..., xi)` ha possibilità di successo, si sceglie un elemento `xi+1` tra le diverse alternative;
- Se con tale scelta si arriva a una soluzione `sol`, quella è la soluzione. Altrimenti si sceglie un diverso `xi+1`;
- Se non si trova una soluzione dopo aver esaminato tutte le possibilità: `fallimento` ;
- Se si verifica che `(x1, ..., xi)` non ha possibilità di successo, non si generano tutte le sequenze `(x1, ..., xi , ....)` .

Così facendo riusciamo a risparmiare un notevole tempo computazionale.

Capitolo 2

Realizzazione

Il codice può essere visto come l'insieme di 3 sezioni differenti:

- Definizione delle eccezioni (vedi Sezione 2.1);
- Funzione per la generazione randomica di stringhe e relative funzioni ausiliarie. Sono state create originariamente come funzioni esterne al solo scopo di semplificare la fase di testing del programma stesso ma vista la loro utilità ho deciso di incorporarle al progetto (vedi Sezione 2.2);
- Funzione per la ricerca di substring, tra tutte le stringhe date in input, e relative funzioni ausiliarie (vedi Sezione 2.4.2).

2.1 Definizione eccezioni

Nella prima parte del codice troviamo la definizione delle eccezioni. In totale sono 9, divise per gruppi in base alle funzioni principali in cui sono utilizzate:

```
(* Eccezioni - genera_lista() *)
exception NumeroStringheNonValido of string;;
exception LunghezzaMinimaNonValida of string;;
exception LunghezzaMassimaNonValida of string;;

(* Eccezioni - continua_ricerca() *)
exception ProlungaRicerca;;
exception ProlungaRicercaCambioK;;
exception ProlungaRicercaIncrementoK;;

(* Eccezioni - ricerca_substring_helper() *)
exception NonPresente;;
exception GiaInRisultato;;

(* Eccezioni - ricerca_substring() *)
exception KNonUtilizzabile of string;;
exception ListaVuota of string;;
```

2.1.1 Eccezioni per la funzione genera_lista

Come possiamo capire dal nome della funzione, tali eccezioni vengono usate dalle funzioni che si occupano della generazione delle stringhe randomiche. In particolare abbiamo:

- exception **NumeroStringheNonValido** of string: viene sollevata quando il numero di stringhe da generare è uguale o minore di 0;
- exception **LunghezzaMinimaNonValida** of string: viene sollevata quando il valore di lunghezza minimo per le stringhe da generare è uguale o minore di 0;
- exception **LunghezzaMassimaNonValida** of string: viene sollevata quando il valore di lunghezza massima per le stringhe da generare è minore del valore di lunghezza minima impostato precedentemente.

2.1.2 Eccezioni per la funzione continua_ricerca

Come possiamo capire dal titolo, tali eccezioni vengono usate dalle funzioni che si occupano della richiesta, all'utente, di proseguire la ricerca di ulteriori substring oltre a quella appena trovata. In particolare abbiamo:

- exception **ProlungaRicerca**: viene sollevata quando l'utente vuole proseguire la ricerca mantenendo la stessa lunghezza (K) della substring da cercare;
- exception **ProlungaRicercaCambioK**: viene sollevata quando l'utente vuole proseguire la ricerca ma cambiando il valore della lunghezza (K) della substring da cercare;
- exception **ProlungaRicercaIncrementoK**: viene sollevata quando l'utente vuole proseguire la ricerca ma incrementando di 1 il valore della lunghezza (K) della substring da cercare.

2.1.3 Eccezioni per la funzione ricerca_substring_helper

Come possiamo capire dal titolo, tali eccezioni vengono usate dalla funzione ausiliaria di ricerca substring. In particolare abbiamo:

- exception **NonPresente**: viene sollevata quando la substring che è appena stata ricercata non compare all'interno di una stringa, causando quindi l'abbandono del path di ricerca soluzione;
- exception **GiaInRisultato**: viene sollevata quando la substring di lunghezza K che stiamo per ricercare è già stata trovata come soluzione in precedenza. Bypassiamo quindi la ricerca a partire da quell'index, spostandoci al successivo, se disponibile.

2.1.4 Eccezioni per la funzione ricerca_substring

Come possiamo capire dal nome della funzione, tali eccezioni vengono usate dalla principale funzione di ricerca substring. In particolare abbiamo:

- exception **KNonUtilizzabile**: viene sollevata in due occasioni, ovvero:
  - quando il valore della lunghezza (K) della substring da cercare è uguale o minore di 0;
  - quando il valore della lunghezza (K) della substring da cercare è maggiore della lunghezza della stringa più corta presente nella lista di stringhe data in input alla funzione.
- exception **ListaVuota** of string: viene sollevata quando la lista data in input alla funzione non contiene alcun elemento.

2.2 Generazione stringhe randomiche

Tale sezione contiene una serie di funzioni di cui una principale e le restanti ausiliarie. Il suo scopo è quello di generare una lista di stringhe, che viene poi salvata nel file "*lista.txt*", in base ad alcuni parametri che vengono chiesti interattivamente all'utente, quali:

- numero di stringhe da generare;
- lunghezza minima stringhe da generare;
- lunghezza massima stringhe da generare;
- valore della substring che avranno in comune tutte le stringhe generate.

2.2.1 Funzioni ausiliarie

2.2.1.1 genera_stringa_random

É la funzione ausiliaria di `genera_stringa`. É essa che si occupa effettivamente della generazione randomica di stringhe anche se, per come ho implementato il tutto, dobbiamo richiamare due volte tale funzione per creare un'unica stringa. Il motivo lo troviamo nella Sezione 2.2.1.2 .

```
let rec genera_stringa_random lunghezza risultato =
  if lunghezza <= 0
    then risultato
    (* Char 97 = a,  Char (97+25) = z                *)
    (* Quindi con Random.int 26 riesco a generare un *)
    (* char (in minuscolo) appartenente all'alfabeto *)
    else genera_stringa_random (lunghezza-1) (risultato^(Printf.sprintf "%c"
     (Char.chr (97 + (Random.int 26)))));;
```

2.2.1.2 genera_stringa

É la funzione ausiliaria di `genera_lista` dalla quale viene richiamata passando in input la lunghezza totale della stringa che vogliamo generare e la substring che vogliamo inserire, eventualmente, all'interno di tale stringa. Restituisce in output una stringa che viene generata dalla concatenazione di 3 elementi:

1. output funzione `genera_stringa_random len` che restituisce una stringa randomica di lunghezza `len` scelta casualmente tra 0 e un carattere in meno rispetto la lunghezza massima della stringa da generare. Tale lunghezza viene a sua volta scelta casualmente all'interno di un range di valori che vengono impostati dall'utente nella funzione `genera_lista` (vedi Sezione 2.2.2);

2. la substring che vogliamo sia contenuta dalla stringa;

3. una stringa randomica, generata come descritto nel punto 1 ma con lunghezza pari alla lunghezza che deve avere la stringa finale meno la lunghezza della stringa già generata al punto 1.

```
let genera_stringa lunghezza_stringa substring_comune file
    bool_ultima_stringa =
  let lunghezza_base = Random.int lunghezza_stringa in
    let risultato = (genera_stringa_random lunghezza_base "") ^
    substring_comune ^ (genera_stringa_random (lunghezza_stringa -
    lunghezza_base) "") in
      if (not bool_ultima_stringa)
        then Printf.fprintf file "%S; " risultato
        else Printf.fprintf file "%S]" risultato;
      risultato ;;
```

2.2.2 Funzione principale - genera_lista

Come dice il titolo, questa è la funzione principale che tramite il richiamo delle due funzioni ausiliarie `genera_stringa_random` (vedi Sezione 2.2.1.1) e `genera_stringa` (vedi Sezione 2.2.1.2) permette la creazione di una lista di stringhe tutte generate casualmente, con lunghezze comprese tra i valori minimo e massimo scelti interattivamente dall'utente proprio in questa funzione.

```
let genera_lista () =
  print_string("\nQuante stringhe vuoi generare? ");
  let numero_stringhe = read_int() in
  if numero_stringhe <= 0
    then raise (NumeroStringheNonValido "Il numero delle strighe da generare deve essere > 0")
    else (
      print_string("\nInserisci la substring che tutte le stringhe devono avere in comune: ");
      let substring_comune = read_line() in
      print_string("\nInserisci il valore di lunghezza minimo per le stringhe da generare: ");
      let lunghezza_min = read_int() in
      if lunghezza_min <= 0
        then raise (LunghezzaMinimaNonValida "Il valore di lunghezza minimo per le stringhe da generare deve essere > 0")
        else (
          print_string("\nInserisci il valore di lunghezza massima per le stringhe da generare (ad esso verra' aggiunto il valore della lunghezza della substring in comune): ");
          let lunghezza_max = read_int() in
          if lunghezza_max < lunghezza_min
            then raise (LunghezzaMassimaNonValida "Il valore di lunghezza massima per le stringhe da generare deve essere >= del valore di lunghezza minimo")
            else (
              let file = open_out "lista.txt" in
              Printf.fprintf file "[";
              let risultato =
              let rec genera_lista_helper numero_stringhe lista_risultato =
              if numero_stringhe = 0
                then lista_risultato
                else (
                  let lunghezza_random = (lunghezza_min + Random.int (lunghezza_max - lunghezza_min + 1)) in
                  genera_lista_helper (numero_stringhe-1) ((genera_stringa lunghezza_random substring_comune file (if numero_stringhe = 1 then true else false)) :: lista_risultato)
              )
```

```
29              in genera_lista_helper numero_stringhe [];
30              in close_out file;
31              print_string("\nLista salvata in \"lista.txt\"\n");
32              risultato
33            )
34        )
35    );;
```

2.3 Continua ricerca

Tale sezione contiene una serie di funzioni di cui una principale e due ausiliarie. Il suo scopo è quello di chiedere all'utente se, dopo aver già trovato una "substring soluzione", desidera procedere alla ricerca di un ulteriore substring.

2.3.1 Funzioni ausiliarie

2.3.1.1 stampa_lista

É la funzione ausiliaria di varie funzioni tra cui `stampa_soluzione`. Il suo compito è semplicemente quello di andare a stampare una lista comprendendo quelle che sono le parentesi quadre e i doppi apici per ogni stringa.

```
let stampa_lista lista =
  print_string("[");
  let rec stampa_lista_helper lista = match lista with
    [] -> print_string("]"); ()
    | hd::tl -> Printf.printf "%S" hd;
      if tl <> []
        then print_string("; ");
      stampa_lista_helper tl
in stampa_lista_helper lista;;
```

2.3.1.2 stampa_soluzione

É la funzione ausiliaria di `continua_ricerca`. Il suo compito è quello di stampare l'attuale "substring soluzione" trovata e richiedere all'utente se vuole continuare la ricerca.

```
let stampa_soluzione (res_bool, res_string) lista_risultati richiesta =
  Printf.printf "(%B, %S)\n" res_bool res_string;
  print_string("Lista substring trovate: "); stampa_lista lista_risultati;
  Printf.printf "\n%s\n" richiesta;;
```

2.3.2 Funzione principale - continua_ricerca

Come dice il titolo, questa è la funzione principale che tramite il richiamo della funzione `stampa_soluzione` (vedi Sezione 2.3.1.2) e l'uso di 3 differenti eccezioni (precedentemente descritte nella Sezione 2.1.2), permette di capire se l'utente vuole:

- proseguire la ricerca: lancia un'eccezione tra le seguenti:
  - **ProlungaRicerca** -> per mantenere la stessa K;
  - **ProlungaRicercaIncrementoK** -> per incrementare K di 1;
  - **ProlungaRicercaCambioK** -> per cambiare il valore di K.

  che vengono poi gestite all'interno della funzione `ricerca_substring_helper` (vedi Sezione2.4.1.7).
- interrompere la ricerca: restituisce in output l'ultima soluzione trovata.

```
let continua_ricerca soluzione lista_risultati k =
  stampa_soluzione soluzione lista_risultati "\n\nCercare un'altra substring
     (S/n) ? ";
  let scelta = read_line() in
    if scelta <> "" && (scelta.[0]='n' || scelta.[0]='N')
      then soluzione
      else (
        print_string("\nCosa vuoi fare? (default = [1])\n");
        Printf.printf "[1] Continua con stesso K (K=%d)\n" k;
        Printf.printf "[2] Incrementare il valore di K (diventerebbe K=%d)\n
    " (k+1);
        print_string("[3] Cambiare il valore di K\n\n");
        print_string("Scelta: ");

        let scelta2 = read_line() in
          if scelta2 = "2"
            then raise ProlungaRicercaIncrementoK
            else if scelta2 = "3"
              then raise ProlungaRicercaCambioK
              else raise ProlungaRicerca
      );;
```

2.4 Ricerca substring

Tale sezione contiene una serie di funzioni di cui una principale e le restanti ausiliarie. Il suo scopo è quello di verificare se per tutte le stringhe prese in input, tramite lista, viene trovata una substring in comune di lunghezza K, con K passato in input sempre dall'utente.

2.4.1 Funzioni ausiliarie

2.4.1.1 ordina_ascendente_lunghezza

É la funzione ausiliaria di `ordina_lista`. Essa si occupa del confronto della lunghezza tra stringhe così da permetterne il riordinamento in maniera ascendente. Per farlo restituisce:

- 1: se la lunghezza della prima stringa è maggiore della seconda;
- 0: se la lunghezza della prima stringa è uguale a quello della seconda;
- -1: se la lunghezza della seconda stringa è maggiore della prima;

```
let ordina_ascendente_lunghezza stringa1 stringa2 =
  let lunghezza_s1 = String.length stringa1 in
    let lunghezza_s2 = String.length stringa2 in
      if lunghezza_s1 > lunghezza_s2
        then 1
        else if lunghezza_s1 = lunghezza_s2
          then 0
          else -1;;
```

2.4.1.2 ordina_lista

Tramite il richiamo della funzione ausiliaria `ordina_ascendente_lunghezza` e `List.sort`, tale funzione è in grado di prendere in input una lista di stringhe e restituirla in output come lista di stringhe ordinate per lunghezza ascendente.

```
let ordina_lista lista = List.sort ordina_ascendente_lunghezza lista;;
```

2.4.1.3 uguale

É la funzione ausiliaria di `gia_in_risultato_helper`. Essa è equivalente all'operatore built-in di uguaglianza e quindi verifica se due valori sono uguali o no. Abbiamo dovuto creare questa funzione perchè, come vedremo nella Sezione 2.4.1.4, la funzione `List.exists` richiede a sua volta in input una funzione.

```
let uguale = (=);;
```

2.4.1.4 gia_in_risultato_helper

É la funzione ausiliaria di `gia_in_risultato`. Essa, tramite le funzioni `uguale` e `List.exists`, permette di verificare se una determinata stringa, che nel nostro caso sarà una substring, compare già all'interno di una lista nonchè quella che contiene tutte le "substring soluzione" già trovate.

```
let gia_in_risultato_helper substring lista_risultati = List.exists (
  uguale substring) lista_risultati;;
```

2.4.1.5 gia_in_risultato

É la funzione ausiliaria di `ricerca_substring_helper`. Essa controlla se la substring che stiamo per ricercare è già presente all'interno della lista di substring precedentemente trovate così da non doverla ricercare nuovamente.

```
let gia_in_risultato contatore_char index k stringa_piu_corta
    lista_risultati =
  if (List.length lista_risultati > 0 && (contatore_char = 0 && (index+k) <=
     (String.length stringa_piu_corta)))
    then let substring_completa = (String.sub stringa_piu_corta index k) in
      if (gia_in_risultato_helper substring_completa lista_risultati)
        then (
          print_string("\n** ricerca_substring_helper -- substring \"" ^
    substring_completa ^ "\" gia trovata precedentemente **\n");
          true
        ) else false
    else false;;
```

2.4.1.6 cerca_substring_in_stringa

É la funzione ausiliaria di `ricerca_substring_helper`. Essa verifica se una specifica substring, appartenente alla stringa più corta, è presente all'interno di una specifica stringa tra quelle che compongo la lista di input.

```
let rec cerca_substring_in_stringa substring stringa_confronto i j
    contatore_char k =
  let len_substring = String.length substring in
  let len_stringa_confronto = String.length stringa_confronto in
    (* Continuo a testare una stringa che NON ha ottenuto un match *)
    (* solo fino a lunghezza stringa - k perchè oltre quell'index, *)
    (* anche se trovassi un match di un char, non arriverei ad     *)
    (* avere una substring di lunghezza k                          *)
    if (j <= (len_stringa_confronto-k) || (j < len_stringa_confronto &&
    contatore_char > 0)) (* per testare tutta la stringa:  j <
    len_stringa_confronto *)
      then (
        Printf.printf "i: %d\n" i;
        Printf.printf "j: %d\n" j;
        Printf.printf "        substring.[i]: %C\n" substring.[i];
        Printf.printf "stringa_confronto.[j]: %C\n\n" stringa_confronto.[j];
        (* Printf.printf "len_substring-1: %d\n\n" (len_substring-1); *)

        if substring.[i] = stringa_confronto.[j]
          (* Se ho letto tutta la substring e il numero dei char trovati *)
          (* è uguale alla lunghezza della substring, allora ho trovato  *)
          (* quella substring nella stringa                              *)
          then if (len_substring-1) = contatore_char
            then true
            else cerca_substring_in_stringa substring stringa_confronto (i
    +1) (j+1) (contatore_char+1) k
          else cerca_substring_in_stringa substring stringa_confronto 0 (j+1
     -contatore_char) 0 k
      )
      else false;;
```

Come possiamo notare dai commenti nel codice sopra riportato, per risparmiare del tempo di computazione sono andato a limitare il test delle stringhe che NON hanno ottenuto un match solo fino a `lunghezza stringa - k` perchè oltre quell'index, anche se trovassi un match di un char, non arriverei ad avere una substring di lunghezza k.

2.4.1.7 ricerca_substring_helper

É la funzione ausiliaria di `ricerca_substring`. Utilizza il **backtracking** e permette di confrontare la stringa più corta con il resto delle stringhe presenti nella lista di input al fine di verificare la presenza di una substring di lunghezza **K** in comune tra tutte esse.

```
let rec ricerca_substring_helper stringa_piu_corta lista k index
    contatore_char lista_risultati =
  let len_stringa_piu_corta = String.length stringa_piu_corta in
  (* Continuo a testare la stringa più corta che NON ha ottenuto un match *)
  (* solo fino a lunghezza stringa - k perchè oltre quell'index, anche se *)
  (* trovassi un match di un char, non arriverei ad avere una substring   *)
  (* di lunghezza k                                                       *)
  if (((index+contatore_char) < len_stringa_piu_corta  && contatore_char >
    0) || index <= (len_stringa_piu_corta-k))
    then
      try
        if (gia_in_risultato contatore_char index k stringa_piu_corta
    lista_risultati)
          then raise GiaInRisultato
          else (
            let rec cerca_substring_in_lista lista_da_testare = match
    lista_da_testare with
              (* Guardo se la lunghezza della substring appena trovata è *)
              (* inferiore a K e in caso affermativo continuo la ricerca *)
              (* sull'attuale substring concatenata al char successivo   *)
              [] -> if (contatore_char+1) < k
                then (
                  print_string("** ricerca_substring_helper -- testate tutte
     le stringhe **\n");
                  Printf.printf "index:          %d\n" index;
                  Printf.printf "contatore_char: %d\n\n" contatore_char;
                  ricerca_substring_helper stringa_piu_corta lista k index (
    contatore_char+1) lista_risultati
                )
                (* Ho trovato una soluzione, ovvero *)
                (* una substring di lunghezza K     *)
                else (
                  (* Chiedo all'utente se vuole continuare la ricerca *)
                  (* di altre substring, magari cambiando anche K     *)
                  let risultato = String.sub stringa_piu_corta index (
    contatore_char+1) in
                  let lista_risultati_aggiornata = (risultato ::
    lista_risultati) in
                    try
                      continua_ricerca (true, risultato)
    lista_risultati_aggiornata k
                    with
```

```
34                      | ProlungaRicerca -> ricerca_substring_helper
    stringa_piu_corta lista k (index+1) 0 lista_risultati_aggiornata
35                      | ProlungaRicercaIncrementoK -> if k > String.length
    stringa_piu_corta
36                        then let eccezione = ("Usare un K <= lunghezza
    stringa piu' corta nella lista (in questo caso max K = "^(string_of_int
    (String.length stringa_piu_corta))^")") in
37                        raise (KNonUtilizzabile eccezione)
38                        else ricerca_substring_helper stringa_piu_corta
    lista (k+1) 0 0 []
39                      | ProlungaRicercaCambioK -> print_string("\nInserisci
    il nuovo valore di K: ");
40                        let new_k = read_int() in
41                        if new_k <= 0
42                          then raise (KNonUtilizzabile "Usare un K >= 0")
43                          else if new_k > String.length stringa_piu_corta
44                            then let eccezione = ("Usare un K <= lunghezza
    stringa piu' corta nella lista (in questo caso max K = "^(string_of_int
    (String.length stringa_piu_corta))^")") in
45                            raise (KNonUtilizzabile eccezione)
46                            else ricerca_substring_helper stringa_piu_corta
    lista new_k 0 0 []
47              )
48            | x::rest -> print_string("** ricerca_substring_helper -- test
     stringa successiva **\n");
49                Printf.printf "index:           %d\n" index;
50                Printf.printf "contatore_char: %d\n" contatore_char;
51                Printf.printf "  stringa: %S\n" x;
52                Printf.printf "substring: %S\n\n" (String.sub
    stringa_piu_corta index (contatore_char+1));
53
54                (* Se trovo la substring all'interno dell'attuale   *)
55                (* stringa, proseguo la ricerca nella stringa       *)
56                (* successiva  altrimenti lancio l'eccezione        *)
57                (* NonPresente che va ad effettuare il backtracking *)
58                (* vero e proprio abbandonando questo ramo di       *)
59                (* soluzione e richiamando  l'algoritmo di ricerca  *)
60                (* sull'indice successivo                           *)
61                if (cerca_substring_in_stringa (String.sub stringa_piu_corta
     index (contatore_char+1)) x 0 0 0 k)
62                  then cerca_substring_in_lista rest
63                  else raise NonPresente
64            in cerca_substring_in_lista lista
65          )
66        with
67          (* Backtracking e richiamo algoritmo su indice *)
68          (* successivo della stringa più corta          *)
69          | NonPresente -> print_string("** ricerca_substring_helper --
    testata stringa, fallimento **\n");
70          Printf.printf "index:           %d\n" index;
```

```
71            Printf.printf "contatore_char: %d\n\n" contatore_char;
72
73            ricerca_substring_helper stringa_piu_corta lista k (index+1) 0
    lista_risultati
74
75        (* Sto per ricercare una substring che è già stata trovata come *)
76        (* soluzione quindi salto direttamente all'indice successivo    *)
77        | GiaInRisultato -> Printf.printf "index: %d\n\n" index;
78          ricerca_substring_helper stringa_piu_corta lista k (index+1) 0
    lista_risultati
79
80    else (false, "");;
```

Come possiamo vedere e capire dal codice appena riportato, il funzionamento di tale funzione si basa su una serie di fattori quali:

- valori che gli passiamo in input, ovvero:
  - **stringa_piu_corta**: è la stringa più corta che troviamo all'interno della lista che ci viene passata in input dall'utente con il richiamo della funzione `ricerca_substring`;
  - **lista**: corrisponde alla lista che ci viene passata in input dall'utente con il richiamo della funzione `ricerca_substring` meno la stringa più corta che vi era al suo interno;
  - **k**: lunghezza della substring da ricercare
  - **index**: indice di scansione per la ricerca di substring rispetto la stringa più corta;
  - **contatore_char**: indica la lunghezza della substring che abbiamo già trovato in comune tra tutte le stringhe;
  - **lista_risultati**: contiene l'elenco di tutte le substring, di stessa lunghezza K, che abbiamo trovato fino ad ora.
- utilizzo della ricorsione di coda che ci permette di richiamare la funzione stessa in base al caso di utilizzo ovvero:
  - nel caso in cui non troviamo la substring ricercata all'interno anche di una sola stringa nella lista, incrementiamo l'indice dal quale ricercare le substring all'interno della stringa più corta e azzeriamo il contatore di char trovati;

- nel caso in cui troviamo la substring, di lunghezza minore K, ricercata all'interno di tutte le stringhe nella lista, incrementiamo il contatore di caratteri trovati in comune;
- in caso abbiamo trovato una "substring risultato" ma vogliamo continuare la ricerca, in base al valore che scegliamo per K ci troviamo in un caso differente, ovvero:
  * manteniamo stesso K: andiamo a incrementare l'indice dal quale ricercare le substring all'interno della stringa più corta, azzeriamo il contatore di char e lasciamo tutti i restanti valori identici;
  * incrementiamo K di 1: andiamo a resettare i valori di indice, contatore_char e lista_risultati, lasciamo invariati stringa_piu_corta e lista, e incrementiamo il valore di K;
  * cambiamo valore di K: stessa cosa di quando lo incrementiamo di 1 ma andiamo a modificare il valore di K con quello scelto dall'utente.
- in caso abbiamo prolungato la ricerca delle substring dopo aver già trovato un risultato, se la substring che stiamo per ricercare compare già nella lista delle soluzioni allora incrementiamo l'indice dal quale ricercare le substring all'interno della stringa più corta e azzeriamo il contatore di char trovati.

- utilizzo di una funzione, ricorsiva, interna nominata `cerca_substring_in_lista` che permette di applicare la ricerca della substring ai singoli elementi della lista tramite il **pattern matching**;
- utilizzo e gestione delle eccezioni per poter offrire all'utente la possibilità di continuare la ricerca di altre substring oltre a quella appena trovata.

2.4.2 Funzione principale - ricerca_substring

Come dice il titolo, questa è la funzione principale che tramite il richiamo delle funzioni `ordina_lista` (vedi Sezione 2.4.1.2), `List.tl` e `List.hd` permette di ricavare dalla lista di stringhe prese in input, tutta una serie di valori che verranno poi passati in input nel richiamo della funzione ausiliare `ricerca_substring_helper` (vedi Sezione 2.4.1.7) con la quale andremo effettivamente alla ricerca delle substring di lunghezza K in comune tra tutte le stringhe ricevute.

```
let ricerca_substring lista k =
  if k <= 0
    then raise (KNonUtilizzabile "Usare un K > 0")
    else if lista = []
      then raise (ListaVuota "Usare una lista con almeno un elemento")
      else if List.length lista = 1
        then (false, "")
        else
          let lista_ordinata = ordina_lista lista in
          let stringa_piu_corta = List.hd lista_ordinata in
          let coda = List.tl lista_ordinata in
            if k > String.length stringa_piu_corta
              then let eccezione = ("Usare un K <= lunghezza stringa piu'
    corta nella lista (in questo caso max K = "^(string_of_int (String.
    length stringa_piu_corta))^")") in
              raise (KNonUtilizzabile eccezione)
              else if stringa_piu_corta = ""
                then (false, "")
                else (
                  print_string("\n** Inizio ricerca substring **\n\n");
                  Printf.printf "Parola campione: %S\n" stringa_piu_corta;
                  print_string("Lista in cui cercare la substring: ");
    stampa_lista coda; print_newline();
                  Printf.printf "Lunghezza substring (K): %d\n\n\n" k;
                  ricerca_substring_helper stringa_piu_corta coda k 0 0 []
                );;
```

Come possiamo notare dal codice sopra riportato, tale funzione va inoltre ad effettuare dei controlli di sicurezza per verificare che:

- la lunghezza (K) della substring da cercare sia `>= 0` ;
- la lunghezza (K) della substring da cercare sia `<=` lunghezza stringa più corta all'interno della lista in input;
- la lista in input contenga almeno un elemento.

La funzione restituisce in output una tupla (**bool** * **string**) in cui il valore bool ci indica se la ricerca ha avuto successo o meno mentre il valore string ci ritorna "" se la ricerca non ha avuto successo, altrimenti il valore della substring, di lunghezza K, in comune tra tutte le stringhe.

Capitolo 3

Test e Risultati

Di seguito porto alcuni esempi di test del programma. In particolare nei primi andrò a mostrare il funzionamento generale dell'algoritmo di ricerca substring creando degli esempi ad hoc che mostrino solo determinate caratteristiche di ricerca alla volta. Andrò infine a mostrare un esempio in cui usiamo anche le funzioni di generazione lista.

3.1 Esempio 1

In questo esempio diamo in input alla funzione `ricerca_substring` i seguenti valori:

- `lista = ["abcdef";"ghijabklcdemn";"opqabcders"]`
- `k = 3`

Per prima cosa l'algoritmo andrà ad effettuare dei test di sicurezza sui valori passati in input ed in caso non ci siano problemi andrà a riordinare la lista. In questo caso ci verrà quindi restituita:

```
lista = ["abcdef";"opqabcders";"ghijabklcdemn"]
```

Da essa viene poi estratto il primo elemento nonchè la stringa più corta al suo interno, ovvero `"abcdef"` . Essa verrà usata come stringa base alla quale applicare la ricerca di substring. É stata fatta questa scelta perchè ovviamente ci permette di minimizzare i tempi di ricerca essendo che le substring

che non sono in essa non potranno essere un risultato dell'algoritmo, anche se comparissero in tutte le altre stringhe. Il resto della lista viene invece preso come "lista di confronto". Il tutto ci viene poi mostrato a schermo per una maggiore sicurezza sui dati effettivi che verranno usati per la ricerca delle substring:

```
# ricerca_substring ["abcdef";"ghijabklcdemn";"opqabcders"] 3;;

** Inizio ricerca substring **

Parola campione: "abcdef"
Lista in cui cercare la substring: ["opqabcders"; "ghijabklcdemn"]
Lunghezza substring (K): 3
```

Subito dopo inizia la ricerca vera e propria grazie al richiamo della funzione `ricerca_substring_helper` :

```
** ricerca_substring_helper -- test stringa successiva **
index:          0
contatore_char: 0
  stringa: "opqabcders"
substring: "a"

i: 0
j: 0
        substring.[i]: 'a'
stringa_confronto.[j]: 'o'

i: 0
j: 1
        substring.[i]: 'a'
stringa_confronto.[j]: 'p'

i: 0
j: 2
        substring.[i]: 'a'
stringa_confronto.[j]: 'q'

i: 0
j: 3
        substring.[i]: 'a'
stringa_confronto.[j]: 'a'
```

Figura 3.1: Inizio ricerca a indice e contatore 0.

Come possiamo vedere dalla Figura 3.1, inizialmente sia l'indice di ricerca (`index`) che il contatore char trovati (`contatore_char`) sono posti a `0` e come

prima stringa viene preso il primo elemento della lista di confronto, ovvero `"opqabcders"`. Andiamo quindi a confrontare se l'indice (`i=0 -> 'a'`) della stringa base `"abcdef"` viene trovato all'interno della stringa `"opqabcders"` partendo dal relativo indice `j=0 -> 'o'`.

Dopo alcuni tentativi, arrivati all'indice `j=3 -> 'a'` si ottiene un riscontro e quindi si procede con la ricerca della stessa substring `"a"` all'interno del secondo elemento della lista, ovvero `"ghijabklcdemn"`. Anche in questo caso, come possiamo vedere in Figura 3.2,troviamo un riscontro infatti l'indice `j=4 -> 'a'` .

```
** ricerca_substring_helper -- test stringa successiva **
index:          0
contatore_char: 0
  stringa: "ghijabklcdemn"
substring: "a"

i: 0
j: 0
        substring.[i]: 'a'
stringa_confronto.[j]: 'g'

i: 0
j: 1
        substring.[i]: 'a'
stringa_confronto.[j]: 'h'

i: 0
j: 2
        substring.[i]: 'a'
stringa_confronto.[j]: 'i'

i: 0
j: 3
        substring.[i]: 'a'
stringa_confronto.[j]: 'j'

i: 0
j: 4
        substring.[i]: 'a'
stringa_confronto.[j]: 'a'
```

Figura 3.2: Ricerca substring `"a"` con indice e contatore resettati per nuova stringa.

Essendo che la lista di stringhe è terminata, si va a verificare se la lun-

ghezza della substring che abbiamo appena trovato in comune è uguale a K. Se si allora la ricerca viene interrotta, altrimenti si continua confrontando nuovamente tutta la lista di stringhe con una nuova substring data dalla precedente concatenata al rispettivo char successivo. In questo caso `String.length "a" = 1` e `K = 3` quindi dobbiamo continuare la ricerca richiamando la funzione `ricerca_substring_helper` con il `contatore_char` incrementato di 1 così che la substring ricercata non sarà più `"a"` ma sarà `"ab"` in quanto trovata come:

```
String.sub stringa_piu_corta index (contatore_char+1)
```

Di seguito non mostro le foto ma usando la stessa logica appena vista, capiamo come anche la substring `"ab"` venga trovata in entrambe le stringhe. Cosa diversa per la substring successiva, `"abc"` che invece, come possiamo vedere in Figura 3.3, viene si trovata all'interno di `"opqabcders"`, ma non in `"ghijabklcdemn"` nonostante venga scandita l'intera stringa.

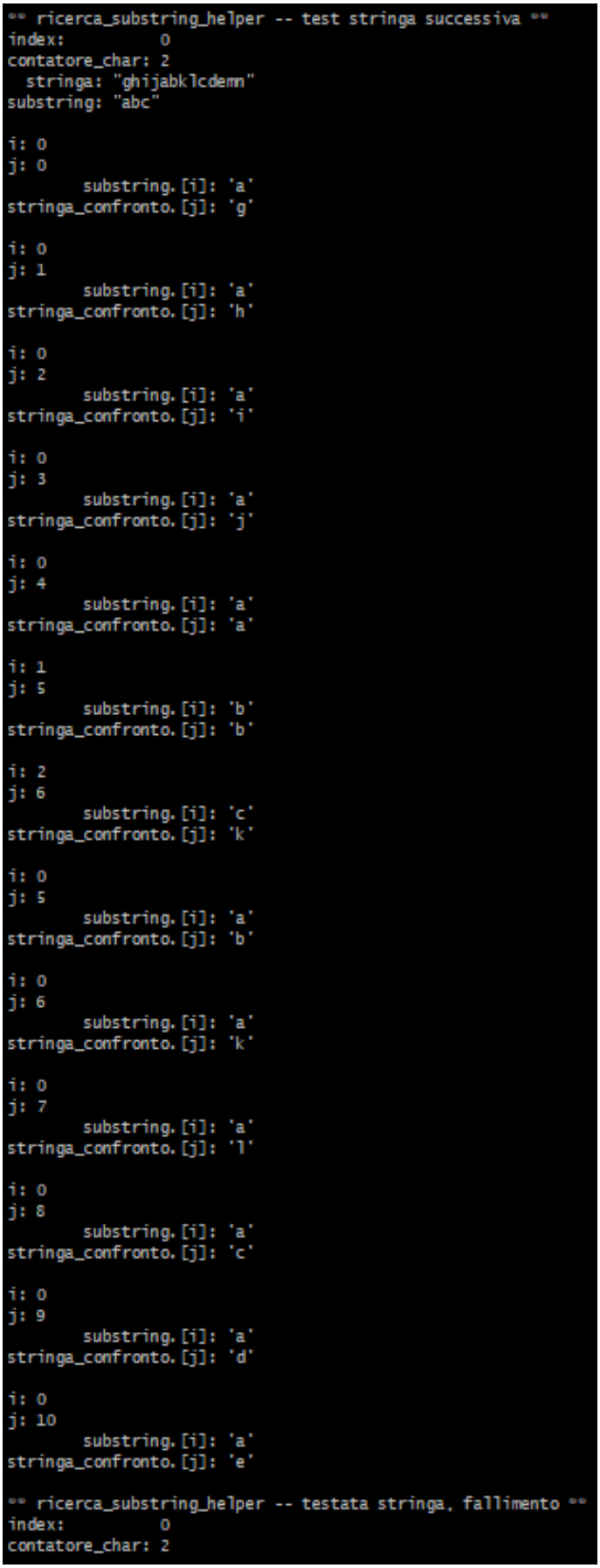

Figura 3.3: Ricerca substring `"abc"` in `"ghijabklcdemn"`.

Ciò ci permette di capire che questo path di ricerca non può avere soluzione e quindi dobbiamo abbandonarlo. Tramite il **backtracking** l'algoritmo capisce che deve tornare indietro nella ricerca delle soluzioni e passare dalla substring `"abc"` a `"b"`.

Applicando nuovamente la stessa logica notiamo come alla fine otteniamo un albero di ricerca come il seguente:

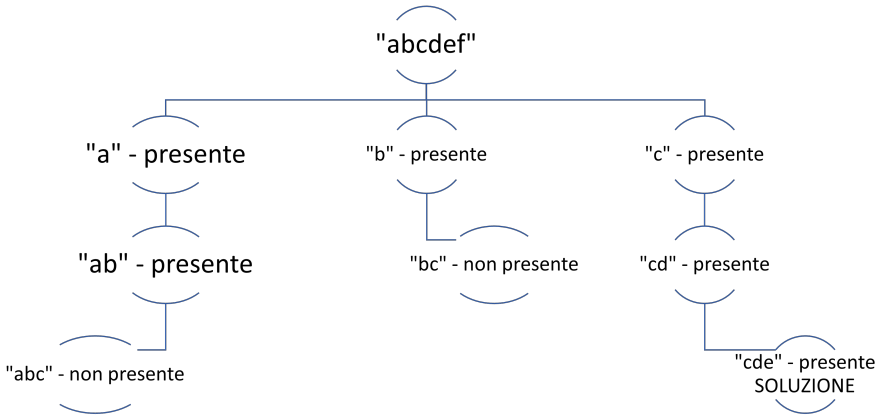

Figura 3.4: Albero di ricerca per substring di lunghezza 3.

dove, come soluzione con `K=3`, riusciamo a trovare la substring `"cde"`.

Dopo aver trovato una soluzione, il programma chiede all'utente se vuole proseguire la ricerca e in caso affermativo gli viene chiesto che valore di K vuole utilizzare o meglio se mantenere lo stesso o se incrementarlo/cambiarlo. In caso negativo viene semplicemente restituito l'ultimo risultato.

```
(true, "cde")
Lista substring trovate: ["cde"]


Cercare un'altra substring (S/n) ?
n
- : bool * string = (true, "cde")
```

Figura 3.5: Risultato ricerca per substring di lunghezza 3.

3.2 Esempio 2

In questo esempio andremo a vedere un caso particolare in cui all'interno della stessa parola compaiono due o più identiche "substring soluzione". Per come sono andato ad implementare l'algoritmo, esso ancor prima di effettuare la ricerca per un nuovo indice va a controllare se prendendo la stringa formata dalla concatenazione con i successivi `k-1` caratteri, va a formare una stringa che è già presente nell'elenco delle soluzioni. In caso affermativo va a bypassare quell'indice e prosegue con la ricerca del successivo.

In questo esempio diamo in input alla funzione `ricerca_substring` i seguenti valori:

- `lista = ["ciccia";"ciccia"]`
- `k = 1`

Essendo che le due stringhe sono identiche l'ordinamento non cambia e quindi procediamo direttamente alla ricerca. Essendo che `K=1`, le stringhe sono solo 2 e sono identiche, si ha che ogni char della stringa può essere considerato una soluzione. Andando ad eseguire l'algoritmo però, come possiamo notare in Figura 3.6, dopo aver già trovato le prime due "substring soluzione" ovvero i primi due char della stringa (`"c"`, `"i"`) e aver richiesto di proseguire la ricerca mantenendo `K=1`, notiamo che ci vengono restituiti 3 messaggi di errore corrispondenti appunto al bypass della ricerca dei 3 caratteri successivi (rispettivamente `"c"`, `"c"`, `"i"`) in quanto già presenti nella lista delle soluzioni. Così facendo quindi, essendo disponibile un'ulteriore char in cui ricercare la soluzione, viene effettuata la ricerca di quest'ultimo (`"a"`) che ovviamente va a formare un'ulteriore elemento di soluzione.

Stessa cosa avverrebbe se K avesse un qualsiasi altro valore. Difatti, se continuiamo la ricerca incrementando il valore di K a 2, in Figura 3.7 possiamo notare lo stesso identico comportamento.

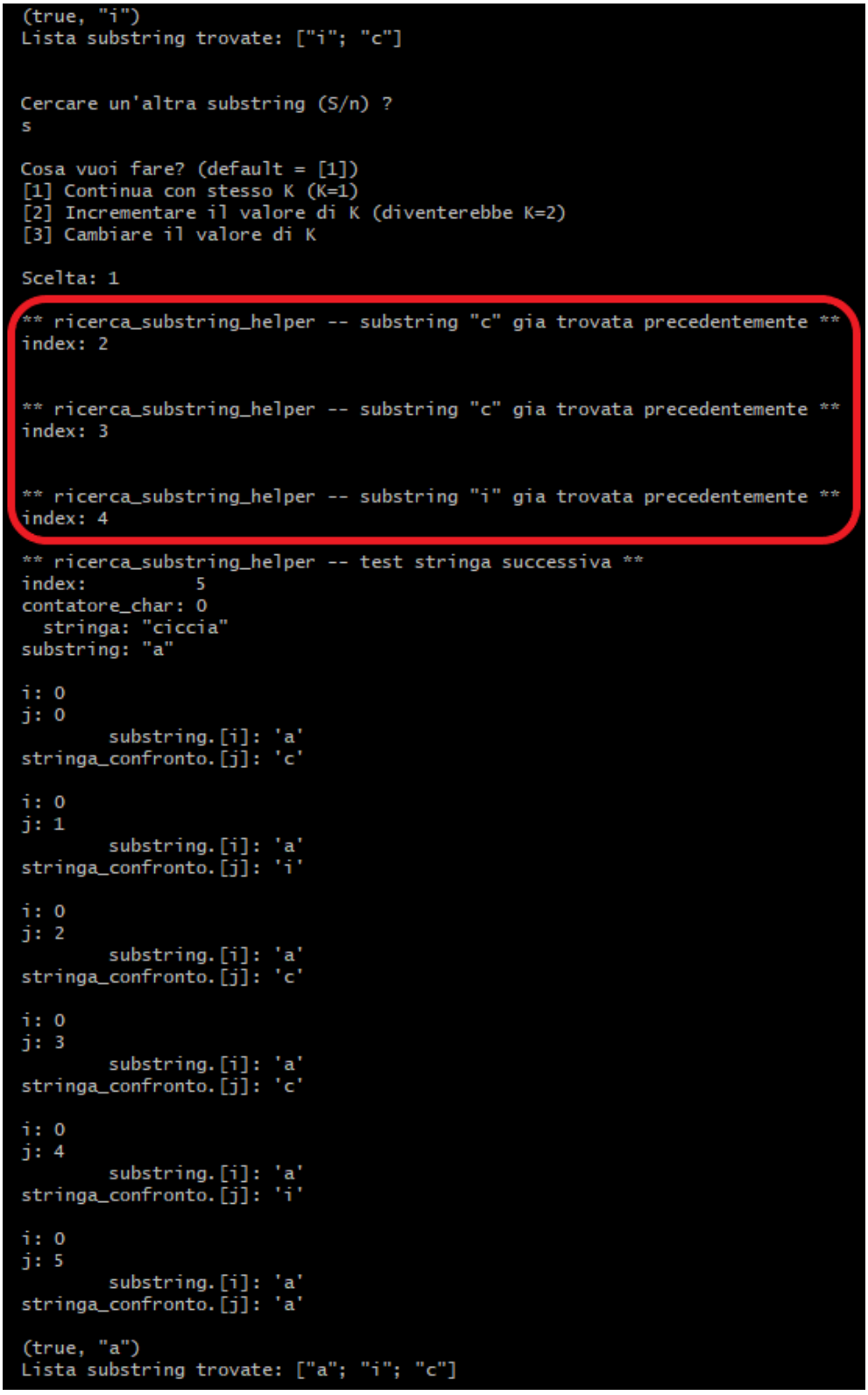

Figura 3.6: Bypass ricerca substring già trovate come soluzione.

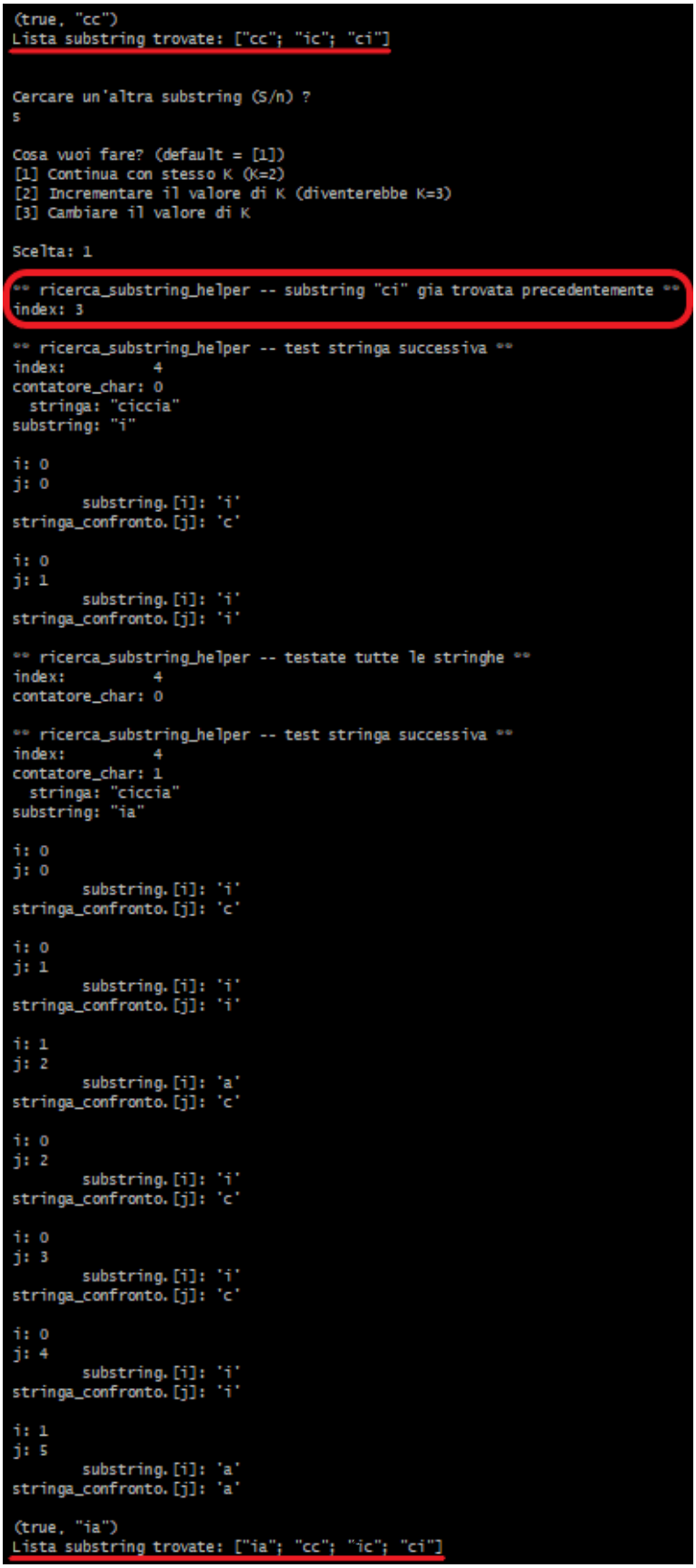

Figura 3.7: Bypass ricerca substring già trovate come soluzione.

3.3 Esempio 3

In questo esempio andremo a vedere il caso in cui usiamo anche le funzioni di generazione lista. In particolare andrò a richiedere in output una lista con:

- lunghezza minima stringhe: 6 char;
- lunghezza massima stringhe: 10 char;
- substring in comune tra tutte le stringhe generate: `"gig"`;
- lunga 500 stringhe.

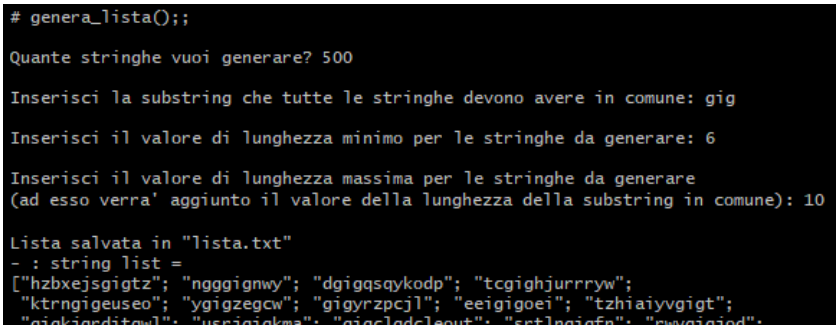

Figura 3.8: Richiesta generazione lista di stringhe.

Eseguendo la funzione `genera_lista` e passando interattivamente i valori riportati sopra, ho ottenuto la seguente lista salvata nel file "*lista.txt*" :

Usando come riferimento i primi elementi della lista in Figura 3.8, possiamo notare come effettivamente ognuno di essi contiene la substring `"gig"` e che in generale hanno delle lunghezze variabili.

Andando poi ad eseguire la funzione `ricerca_substring` passandogli in input quella lista e K pari alla lunghezza della substring in comune inserita precedentemente (`String.length "gig" = 3`), dopo pochissimi secondi otteniamo il seguente risultato:

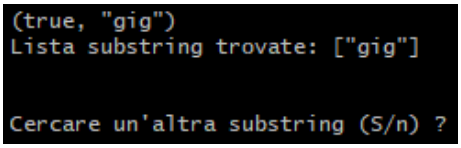

Figura 3.9: Ricerca substring tra 500 stringhe con `K=3`.

Capitolo 4

Conclusioni

Nonostante questo algoritmo venga eseguito in un tempo non troppo elevato, sicuramente un algoritmo in ampiezza invece che in profondità (quale il mio) sarebbe stato molto più performante. Avrei potuto infatti prendere la prima substring di lunghezza K, se presente e se non presente già terminavo la ricerca. Se invece esisteva, cercavo se questa substring compariva nel primo elemento della lista di confronto. Se compariva allora lo ricercavo anche nel secondo elemento della lista e così via. Se ad una certa non trovavo una corrispondenza allora effettuavo backtraking scorrendo di 1 l'index dal quale prendere la substring sulla stringa più corta per poi ripetere la ricerca come visto poco prima. Se ottenevo un riscontro tra tutte le stringhe allora sarebbe stato il risultato altrimenti, dopo aver testato tutte le substring "utili" nella stringa più corta, avrei ritornato false come risultato. Il tutto mi avrebbe quindi risparmiato gli innumerevoli scorrimenti in lettura della lista di confronto che sono attualmente costretto a fare.