Progetto di Algoritmi e Strutture Dati

Michele Lazzeri Matricola 822879

1 Traduzione da Linguaggio di contesto a Linguaggio specifico

I 'nucleotidi' A,T,C,G rappresentano un Alfabeto Σ. Una concatenazione di simboli dell'Alfabeto rappresenta una Stringa fr e nel contesto viene chiamato 'Frammento di DNA'. L'insieme di tutti i 'Frammenti di DNA' ammissibili F rappresenta quindi il Linguaggio.
Un 'enzima' rappresenta una funzione $\epsilon: F \to F$. L'insieme degli enzimi' ε rappresenta quindi una classe di funzioni di questo tipo.
Una 'trasformazione' rappresenta il risultato di un prodotto di funzioni.
Un 'esperimento' rappresenta un insieme di stringhe fr e di enzimi ϵ.
La 'durata' Δ rappresenta il numero massimo di funzioni applicabili per ogni Stringa fr in ogni esperimento.
L"energia richiesta in UEA' rappresenta una funzione peso che associa ad ogni 'enzima' ε un numero reale: $energia(\varepsilon): \epsilon \to \mathbb{R}_+$
Inoltre $\forall$ trasformazione t $= \{e_1, e_2, \ldots, e_k\}$ abbiamo che $energia(t) = \sum_{j=1}^{k} energia(e_j)$

2 Scelta delle strutture adatte

2.1 Frammenti di Dna / Stringhe

Il metodo più semplice per immagazzinare le singole stringhe è quello di utilizzare degli array. Gli array:

- permettono un accesso diretto ad ogni singolo elemento (simbolo);
- richiedono uno spazio di memoria contenuto
- sono semplici da gestire e si adattano bene ad operazioni su stringhe.

Una possibile struttura in C potrebbe essere composta in questo modo:

```
struct stringa {
    nome char[50];
    int lunghezza;
}
```

In questo modo è possibile inserire nuovi elementi alla fine dell'array senza dovere riallocare l'intera array (supponendo di non superare il limite dei 50 caratteri) in tempo O(1), il rovesciamento dell'array può essere effettuato tramite un array di appoggio in tempo O(n), l'eliminazione degli ultimi n elementi in un tempo O(1), l'eliminazione e l'inserimento di un elemento non alla fine dell'array possono comunque essere realizzati shiftando i vari elementi dell'array. La semplicità della struttura permette in ogni caso di implementare le varie funzioni (gli 'enzimi') facilmente.

2.2 Libreria degli enzimi

La libreria degli enzimi raccoglie le varie funzioni disponibili per la manipolazione dei frammenti di DNA. Le due funzioni che devono essere servite da tale libreria sono char *enzima (char *nome_enzima, char *frammento_src) e int energia_enzima (char *nome_enzima). Tali funzioni possono essere implementate tramite un costrutto

```
switch(nome_enzima) {
    case A:
    ...
    case B:
    ...
}
```

oppure utilizzando un albero binario di ricerca. Quest'ultima opzione garantisce i seguenti benefici:

- Tempo di accesso proporzionale alla lunghezza dell'albero: tale tempo nel caso peggiore si rivela essere ugualmente $O(n)$, ma nel caso medio il tempo scende a $O(log(n))$, dove n è il numero di enzimi presenti.
- Tempo di inserimento in qualsiasi posizione: $O(n)$ nel caso peggiore, $O(log(n))$ nel caso medio

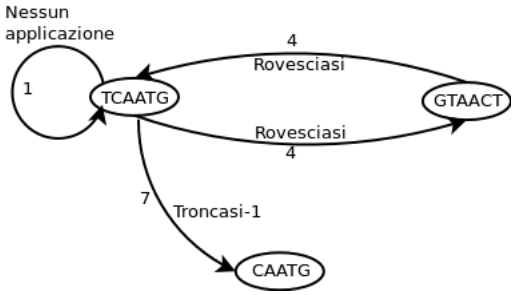
I singoli nodi possono essere strutture formate da un puntatore a funzione, una stringa identificativa e un intero per l'energia richiesta. Tale libreria fornisce le funzioni char *enzima(char *nome-enzima, char *frammento) e int energia-enzima(char *nome-enzima). Per ridurre il tempo a $O(log(n))$ anche nel caso peggiore si potrebbe ricorrere a una struttura come un albero 2-3 o un B-albero, per semplicità verrà però usato un albero di ricerca binario.

2.3 Esperimento

L'intero esperimento può essere rappresentato tramite un grafo ordinato $G =< V, E >$ dove V sono i nodi (in questo caso le varie Stringhe / Frammenti di DNA) e E è un sottoinsieme del prodotto cartesiano $V \times V$ tale che $e = \{x, y\} \in E \iff \exists$ enzima $enz \mid enzima(enz, x) = y$ Convenzioni utilizzate:

- L'energia necessaria per ogni enzima è rappresentata dal peso del lato.
- La possibilità che ad un frammento non si attacchi nessun enzima viene 'emulata' tramite un enzima nullo che restituisce lo stesso frammento. All'interno del grafo tale lato viene raffigurato tramite un cappio.

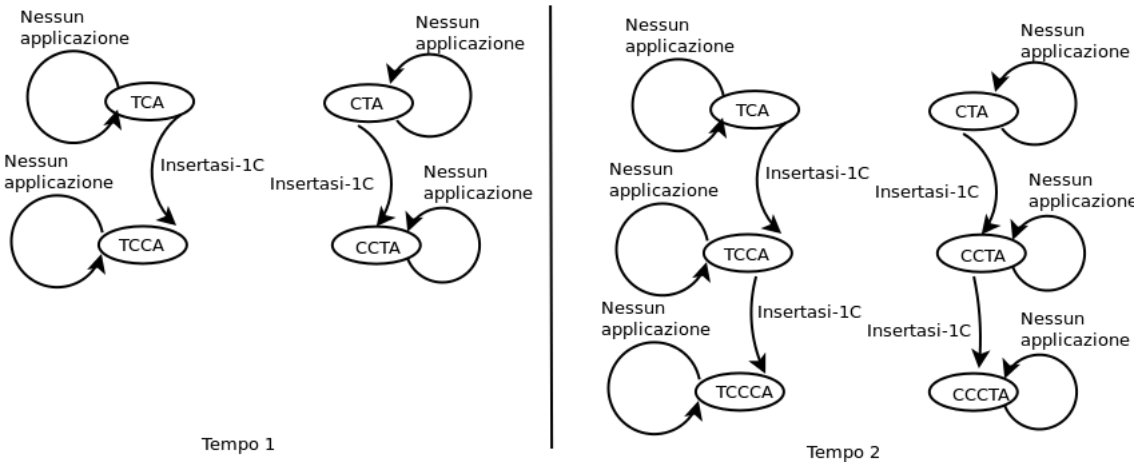
Ad esempio il grafo:



Rappresenta la situazione per cui:

- 'TCAATG' può diventare 'GTAACT' tramite l'enzima Rovesciasi di peso 4(che 'rovescia' i frammenti)
- 'TCAATG' può diventare 'CAATG' tramite l'enzima troncasi-1 di peso 7
- 'GTAACT' può diventare 'TCAATG' tramite l'enzima rovesciasi di peso 4
- 'TCAATG' resta 'TCAATG' senza applicazione di enzimi
- ...

Il tempo determina il numero di lati e di frammenti, infatti partendo dai frammenti {'TCA','CTA'} e dall'enzima Insertasi-1C i grafi al tempo 1 e al tempo 2 sono i seguenti:



L'implementazione del grafo avviene tramite liste di adiacenza, con una variazione nella struttura usata per rappresentare i nodi, come spiegato nella sezione 2.3.2.

2.3.1 Inibizione enzimi

L'inibizione degli enzimi può essere implementata tramite un altro albero di ricerca binaria. Prima dell'avvio dell'esperimento, infatti, la situazione del grafo G è quella di avere un certo numero n di vertici e nessun lato. L'esperimento vero e proprio applica ad ogni vertice tutti gli enzimi contenuti in una struttura che deve essere riempita tramite le chiamate di aggiungi_enzima. Invece di utilizzare una lista per tale struttura risulta comodo utilizzare un albero di ricerca binaria. Tale albero sarà formato da nodi che conterranno sia il nome dell'enzima, sia un valore booleano che rappresenta se l'enzima è attivo o inibito. In questo modo l'inibizione di un enzima avviene cercando all'interno di tale albero l'enzima e portando a false il valore booleano associato. L'utilizzo di un albero di ricerca binaria invece di una lista permette di velocizzare sia il processo di inibizione/riattivazione sia quello di inserimento.
All'interno dei nodi di quest'albero sarebbe inoltre possibile inserire anche l'energia richiesta per l'attivazione di tale enzima: in questo modo, pur utilizzando una quantità maggiore di spazio (i dati relativi all'energia richiesta sono già disponibili nella libreria degli enzimi) rende più veloce l'accesso a tali informazioni. Infatti se n è il numero di enzimi a disposizione del professore e k è il numero di enzimi utilizzati in questo esperimento abbiamo che $1 \leq k \leq n$.

2.3.2 I nodi del grafo

L'algoritmo per la costruzione del grafo, dopo l'inserimento dei frammenti iniziali e degli enzimi ha una struttura generale del tipo

```
for nodo v ∈ V do
    for enzima ϵ ∈ ε do
        z:= enzima(ϵ,v);
        a:= cerca(z);
        if ∃ z in V then
            k:= puntatore al nodo preesistente che contiene z
        else
            crea nodo(z);
            k:= puntatore a tale nodo;
        end
        L(v) := INSERISCI((ϵ,k));
    end
end
```

Dove ε è l'insieme degli enzimi inseriti nell'esperimento, V è l'insieme dei nodi, L(v) è la lista di adiacenza del nodo v.
Data la necessità, dopo l'ottenimento del risultato della funzione enzima(ϵ,stringa), di cercare se tale risultato è già presente nella lista dei nodi (in modo tale da creare un lato dalla stringa di origine al risultato) risulta comodo utilizzare anche per la lista dei nodi un albero binario di ricerca, per ridurre tali tempi di ricerca.
L'implementazione scelta per la creazione del grafo risulta quindi essere formata da una struttura (che verrà chiamata Struct) contenente due puntatori, uno all'albero contenente i frammenti (il grafo orientato vero e proprio)(Albero_Framm) e un altro verso l'albero contenente gli enzimi inseriti nell'esperimento (Albero_Enz)

- l'albero contenente i frammenti avrà dei nodi formati da 5 campi:
  - Il frammento di DNA, implementato come descritto in 2.1;
  - Due puntatori, uno al figlio sinistro e uno al figlio destro
  - Un puntatore alla lista di adiacenza (ogni elemento di tale lista sarà formata da un campo nome, che conterrà il nome dell'enzima, e un puntatore al frammento risultante dall'applicazione dell'enzima sul frammento originario)
  - Un valore che indica se tale nodo è stato visitato o meno (spiegazione in 3.10)
- l'albero contenente gli enzimi inseriti avrà dei nodi formati da 4 campi:
  - Il nome dell'enzima
  - Due puntatori, uno al figlio sinistro e uno al figlio destro
  - Un valore booleano che indica se tale enzima è attivo o meno

Anche per questi due alberi per ridurre il tempo di inserimento/ricerca/cancellazione a $O(log(n))$ anche nel caso peggiore si potrebbe ricorrere a una struttura come un albero 2-3 o un B-albero, ma per semplicità verrà usato un albero di ricerca binario.

3 Funzioni in pseudo-codice

3.1 Crea_Nuovo

Questa prima funzione crea semplicemente la struttura di base e inserisce nell'albero degli enzimi l'enzima nullo (di peso 1) rappresentante la non-interazione tra enzimi e frammenti:

```
A:= Albero Binario;
B:= Albero Binario;
B:= INSERISCI(B,('Nessuna Trasformazione',1));
Struct:= INSERISCI(Struct,(A,B));
return Struct;
```

La procedura richiede un tempo O(1) dovendo semplicemente allocare spazio e inserire al primo posto di un albero senza vertici un nodo.

3.2 Cerca(el,rad)

Questa funzione cerca un nodo el all'interno di un albero binario di ricerca con radice rad, se lo trova restituisce la coppia (1,&nodo), se non lo trova restituisce la coppia (0,&padre). In questo secondo caso l'inserimento non richiederà più di trovare il cammino fino ad una foglia, ma basterà confrontare l'elemento da inserire con padre e inserire quindi l'elemento come figlio destro o sinistro:

```
node:= rad;
if rad = el then
    return 1,node;
else if rad < el then
    if des(rad) ≠ NULL then
        return cerca(el,des(rad))
    else
        return 0,node;
    end
else if rad > frammento then
    if sin(rad) ≠ NULL then
        return cerca(el,sin(rad))
    else
        return 0,node;
    end
end
```

Tale procedura ricorsiva effettua un numero di chiamate proporzionale all'altezza dell'albero. Se l'albero è bilanciato l'altezza è dell'ordine di log(n) dove n è il numero dei nodi, se non lo è l'ordine è quello di n. La procedura richiede quindi un tempo $O(log(n))$ nel caso medio e $O(n)$ nel caso peggiore.

3.3 Nuovo_Frammento(frammento)

Questa procedura chiama inizialmente la procedura cerca, se cerca ritorna 1 allora la procedura si arresta, se ritorna 0 confronta il nodo ritornato da cerca con frammento e inserisce quest'ultimo come figlio sinistro o destro:

```
(a,padre) := cerca(frammento,radice(Albero_Framm));
if a=0 then
    if padre > frammento then
        sin(padre):= frammento
    else
        des(padre):= frammento
    end
end
```

Il costo dell'inserimento risulta quindi tutto attribuito alla procedura cerca ed è quindi nuovamente dell'ordine di $O(n)$ nel caso peggiore e $O(log(n))$ nel caso medio.

3.4 Nuovo_Frammento_vis(frammento)

Questa funzione, molto simile alla precedente, inserisce un frammento, ma con la differenza che nel caso in cui venga inserito un nuovo frammento imposta il campo visitato a 1 e restituisce un puntatore al nodo creato.

```
(a,node) := cerca(frammento,radice(Albero_Framm));
if a=0 then
    if padre > frammento then
        sin(node):= (frammento,1);
        node:= sin(node);
    else
        des(node):= (frammento,1);
        node:= des(node)
    end
end
return node;
```

3.5 Nuovo_Enzima(enzima)

Questa funzione cerca l'enzima all'interno del corrispettivo albero, se lo trova lo attiva, se non lo trova crea un nuovo nodo con il flag attivato

```
(a,padre) := cerca(enzima,radice(Albero_Enz));
if a=0 then
    if padre > enzima then
        sin(padre):= (enzima,1)
    else
        des(padre):= (enzima,1)
    end
else
    padre.attivato:= 1
end
```

3.6 Cancella_enz(nodo)

Questa funzione elimina l'albero degli enzimi partendo dalla radice nodo

```
if sin(nodo) ≠ NULL then
    cancella_enz(sin(nodo))
end
if des(nodo) ≠ NULL then
    cancella_enz(des(nodo))
end
cancella nodo;
```

Il numero delle chiamate è proporzionale al numero degli enzimi.

3.7 Cancella_framm(nodo)

Questa funzione elimina l'albero dei frammenti partendo dalla radice nodo

```
if sin(nodo) ≠ NULL then
    cancella_enz(sin(nodo))
end
if des(nodo) ≠ NULL then
    cancella_enz(des(nodo))
end
for e ∈ L(nodo) do
    cancella e
end
cancella nodo;
```

Le chiamate sono proporzionali al numero totale di archi (per la cancellazione delle liste di adiacenza) e al numero di frammenti.

3.8 Elimina_Enzima(enzima)

Questa funzione cerca nell'albero degli enzimi se è presente l'enzima e se lo è lo inibisce:

```
(a,enzima):= cerca(enzima);
if a=1 then
| enzima.attivo:= 0
end
```

Anche per questo algoritmo la ricerca è la componente che determina il costo in tempo

3.9 Prepara_Esperimento(f_file, e_file)

Questa funzione ricrea la struttura di base e inserisce nei due alberi i frammenti e gli enzimi presi rispettivamente dai file f_file e e_file:

```
if struct esiste then
| cancella_enz();
| cancella_framm();
| cancella struct;
end
nuovo_esperimento();
for f in f_file do
| nuovo_Frammento(f)
end
for e in e_file do
| nuovo_Enzima(e)
end
```

3.10 esp(tempo)

Questa funzione fa agire gli enzimi su ogni frammento e confronta i risultati con i frammenti già presenti, inserisce i nuovi frammenti e crea nelle liste di adiacenza i puntatori per i vari nodi adiacenti:

```
if tempo > 0 then
| for t=1,2...tempo do
| | for f ∈ Albero_frammenti do
| | | if f.visitato:= 1 then
| | | | f.visitato:=0;
| | | end
| | end
| | for f ∈ Albero_frammenti do
| | | if f.visitato=0 then
| | | | for e ∈ Albero_enzimi do
| | | | | if e.attivato=1 then
| | | | | | z:= enzima(e,f);
| | | | | | k:= nuovo_Frammento_vis(z);
| | | | | | L(f):= INSERISCI(L(f),(en,k));
| | | | | end
| | | | end
| | | | f.visitato:= 2;
| | | end
| | end
| end
end
```

Come si può notare, dato che l'inserimento di un nuovo frammento è all'interno di un ciclo for che scansiona tutti gli elementi dell'albero dei frammenti, se il frammento fosse inserito senza utilizzare il flag visitato la funzione andrebbe in Loop: supponendo ad esempio di avere il solo frammento 'TAC' e l'enzima 'Insertasi-1A'

la procedura inserirebbe 'TAAC', il ciclo for prenderebbe 'TAAC' come altro frammento e ad esso applicherebbe nuovamente l'enzima e così via. Creando invece il campo visitato, i frammenti vengono impostati inizialmente a 0 e ogni frammento inserito avrà il flag visitato impostato a 1, non verrà quindi riselezionato dal ciclo for. Inoltre gli elementi a cui sono stati applicati gli enzimi al tempo n non andranno riconsiderati al tempo n+1 e il campo visitato assume il valore 2. 2 significa quindi che il frammento è già stato utilizzato nell'esperimento, 1 che sarà utilizzato nel prossimo 'tempo' e 0 che deve essere utilizzato durante questo 'tempo'.
Questa procedura è una delle più dispendiose in termini di tempo dell'intero progetto, infatti supponendo che gli enzimi creino sempre frammenti non esistenti:
indicando con $|F|$ il numero dei frammenti iniziali e con $|E|$ il numero degli enzimi attivati la procedura richiede:

- al **primo** passo il ciclo for dei frammenti cicla $|F|$ volte e quello interno degli enzimi $|E|$ volte, quindi Nuovo_Frammento_vis viene chiamato $|F| * |E|$ volte
- al **secondo** passo il ciclo for dei frammenti cicla $|F| * |E|$ volte e quello interno degli enzimi $|E|$ volte, quindi Nuovo_Frammento_vis viene chiamato $|F| * |E| * |E|$ volte
- ...
- al **n-esimo** passo il ciclo for dei frammenti cicla $|F| * |E|^{n-1}$ volte e quello interno degli enzimi |E| volte, quindi Nuovo_Frammento_vis viene chiamato $|F| * |E|^n$ volte

Nel caso medio Nuovo_Frammento_vis richiede un tempo $O(log(n))$ e quindi il tempo totale risulta essere

$$\sum_{n=1}^{tempo} (f * e^n * O(log(f * e^{n-1}))$$

da cui

$$f * O(log(f)) \sum_{n=1}^{tempo} (e^n) + f * O(log(e)) \sum_{n=1}^{tempo} ((n-1)(e^n))$$

e quindi

$$T(f, e, tempo) = O(f * log(e) * e^{tempo})$$

3.11 Esperimento(tempo)

Questa funzione chiama esp(tempo) e ne stampa i risultati

---

if *tempo* > = **then**
| esp(tempo)
end
Stampa Albero_Frammenti;

3.12 Dijkstra(V,*funz,s)

Questa funzione, esempio di Algoritmo greedy, data come istanza un grafo orientato pesato, crea tre vettori: il primo vettore C specifica per ogni nodo v il costo del cammino minimo da f a v, il secondo vettore P specifica il nodo precedente nel cammino da f a v, il terzo vettore N contiene stringhe indicanti il nome dell'ultimo lato nel cammino da f a v. Ad ogni passo viene scelto il lato u,v con peso minore e viene inserito nella soluzione a patto che non esista già un cammino con peso minore dalla sorgente a v.

```
D:= HEAP;
D:= INSERT(D,s);
for v ∈ V do
    C(v):= ∞;
    P(v):= NULL
end
C(s):= 0;
while ISEMPTY(D)=0 do
    g:= nodo tale che C(nodo) sia minore;
    D:= DELETE(D,g);
    for u ∈ L(g) ∧ C(g)+(*funz)(u.nome_enzima) < C(u) do
        if C(u)=∞ then
            D:= INSERT(D,u)
        end
        C(u):= C(g)+(*funz)(u.nome_enzima);
        P(u):= g;
        N(u):= u.nome_enzima;
    end
end
return C,P,N;
```

Ricordo che la lista di adiacenza di un vertice è formata da nodi al cui interno è contenuto anche il nome del lato che è il nome dell'enzima. La scelta del nodo tale che C(nodo) sia minore può inoltre avvalersi di una coda di priorità in cui il costo rappresenta la priorità. In questo modo la scelta dell'elemento per cui C(elemento) sia minore può essere fatta in un tempo $O(1)$.

3.13 Enzimi(f,g,ε,tempo

Questa funzione elimina le strutture presenti, ricrea un esperimento, inserisce f come unico frammento, inserisce tutti gli enzimi nell'esperimento, chiama esp(tempo), calcola quindi il cammino minimo da f a g supponendo che gli archi abbiano tutti 1 come peso tramite l'algoritmo di Dijkstra e stampa il risultato

```
if struct esiste then
    cancella_enz();
    cancella_framm();
    cancella struct;
end
nuovo_esperimento();
nuovo_Frammento(f);
for e ∈ ε do
    Nuovo_Enzima(e)
end
esp(tempo);
(C,P,N)=dijkstra(Albero_Framm,*(funzione(Funzione_che_ritorna_1)),f);
if C(g)=∞ then
    Stampa 'Non ci sono cammini da f a g'
else
    S:= STACK;
    while P(g) ≠ NULL do
        S:= INSERT(S,N(g));
        g:= P(g);
    end
    STAMPA S;
end
```

Il tempo di esecuzione di questa funzione dipende dalle due chiamate esp(tempo) e dijkstra(...).

3.14 Energia(f,g,ε',tempo)

Questa funzione fa praticamente la stessa cosa della precedente (cammini minimi) ma passa alla funzione Dijkstra un puntatore alla funzione energia_enzima che restituisce l'ammontare in UEA per ogni enzima.

```
if struct esiste then
    cancella_enz();
    cancella_framm();
    cancella struct;
end
nuovo_esperimento();
nuovo_Frammento(f);
for e ∈ ε' do
    Nuovo_Enzima(e)
end
esp(tempo);
(C,P,N)=dijkstra(rad,*(funzione(energia_enzima)),f);
if C(g)=∞ then
    Stampa 'Non ci sono cammini da f a g'
else
    S:= STACK;
    while P(g) ≠ NULL do
        S:= INSERT(S,N(g));
        g:= P(g);
    end
    STAMPA S;
end
```

3.15 Similaritâ(f,g)

Questa funzione utilizza la programmazione dinamica per stabilire la similarità di due stringhe. In particolare vengono create due matrici, la prima serve per contenere i valori di similarità delle varie sotto-stringhe, la seconda per ritrovare le coppie formatesi. Supponendo di avere due stringhe s1,s2 di dimensione N1,N2 le due matrici avranno dimensione N1+1,N2+1. Passando poi di cella in cella da sinistra a destra, dall'alto verso il basso viene poi fatto un calcolo sul massimo tra tre valori: $M_{j,k} = \max \begin{cases} \begin{cases} M_{j-1,k-1}+1 & \text{se } s1[j]=s2[k] \\ M_{j-1,k-1} & else \end{cases} \\ M_{j-1,k} \\ M_{j,k-1} \end{cases}$

In questo modo si sfrutta la programmazione dinamica, in quanto la risoluzione del problema viene suddivisa in sotto-problemi, che vengono utilizzati più volte per la costruzione del risultato finale.
Per la costruzione del risultato finale l'algoritmo parte dall posizione $T_{N1,N2}$, poi se trova una 'd' controlla se s1[j]=s2[k] se si inserisce tale valore in uno stack, se trova un '1' si sposta verso l'alto, se trova un '2' si sposta verso sinistra. Quando uno dei due valori raggiunge 0 stampa la pila.

```
for i=0,1...N1 do
 | M[0][i]:= 0;
end
for i=0,1...N2 do
 | M[i][0]:= 0;
end
for i=1,2...N2 do
 | for j=1,2...N1 do
 |  | if s2[i-1]=s1[j-1] then
 |  |  | a:=M[i-1][j-1]+1;
 |  | else
 |  |  | a:=M[i-1][j-1]
 |  | end
 |  | b:=M[i-1][j];
 |  | c:=M[i][j-1];
 |  | M[i][j]=MASSIMO(a,b,c);
 |  | if M[i][j]=a then
 |  |  | T[i][j]='d';
 |  | else if M[i][j]=b then
 |  |  | T[i][j]='1';
 |  | else
 |  |  | T[i][j]='2';
 |  | end
 | end
end
S:= STACK;
a:= N1;
b:= N2;
while a>0 ∧ b>0 do
 | if T[a][b]='d' then
 |  | a:=a-1;
 |  | b:=b-1;
 |  | if s1[a]=s2[b] then
 |  |  | S:=INSERISCI(S,s1[a])
 |  | end
 | else if T[a][b]='1' then
 |  | a:=a-1;
 | else
 |  | b:=b-1;
 | end
end
Stampa S;
```

Tempo e spazio sono entrambi dell'ordine di grandezza di $O(n*m)$ dove n e m rappresentano la lunghezza delle due stringhe.