Assignment 1, Bioinformatica

Davide Cozzi, 829827

Capitolo 1

esercizio 1

1.1 Versione 1

Si assumano stringhe su $\Sigma = \{a, c, g, t, A, C, G, T\}$ indicizzate a partire dalla posizione 0, per comodità.
L'algoritmo si divide in due parti:

1. divisione in "token" delle due stringhe. A partire da ogni stringa si ottiene un vettore di sottostringhe di uguali caratteri
2. confronto dei due vettori di sottostringhe

Si procede specificando che:

- $length(X)$ restituisce la lunghezza di X, che sia una stringa o un vettore
- $push(X, Y)$ effettua l'operazione di `push` di Y nel vettore/stringa X
- $string(Y)$ effettua il cast di Y a `string`
- $Y[i, e]$ specifica la sottostringa di Y che va dall'indice i incluso all'indice e incluso
- $lowercase(Y)$ porta Y in *lowercase*

Si ha quindi la prima parte dell'algoritmo, che effettua la divisione in "token".

Esempio 1. *Vediamo un esempio di input e output della funzione* `Splitter`.

- ***input****: "aaataaaggggccccctttttttttttttttcc"*
- ***output****: ["aaa", "t", "aaa", "gggg", "ccccc", "ttttttttttttttt", "cc"]*

Si ha quindi lo pseudocodice:

Algorithm 1 Algoritmo per lo split in "token" delle stringhe

```
function SPLITTER(s)
    result ← [ ]
    n ← length(s)
    last_mismatch ← 0
    if n == 1 then
        push(result, string(s[0]))
        return result
    end if
    tmp ← ""
    for i ← 0 to n do
        if i > 0 and s[i − 1] ≠ s[i] then
            push(result, tmp)
            tmp ← ""
        end if
        push(tmp, s[i])
        if i == (n − 1) then
            push(result, tmp)
        end if
    end for
    return result
end function
```

Una volta ottenuto il vettore dei "token" delle due sequenze in input basta confrontare i due vettori.
Prima di vedere il confronto si specifica che:

- le due sequenze vengono trasformate in *lowercase* per praticità
- si assume che le due sequenze non possono essere stringhe vuote ε

Fatte queste assunzioni si può fare ancora un ultima osservazione prima di procedere con l'algoritmo. Qualora i due vettori prodotti dalla funzione

`Splitter` fossero di lunghezza diversa allora l'algoritmo non potrà in nessun caso dire che ci sia stata un"'infezione" in quanto si assume che non ci siano né *gaps* né *inserimenti*. In altri termini qualora i due vettori siano di diversa cardinalità sicuramente, con queste premesse, non è avvenuta alcuna infezione. Vediamo ora le altre casistiche. Si itera contemporaneamente sull'elemento i-esimo dei due vettori e:

- se i due elementi i-esimi presentano il primo carattere diverso allora, date le premesse, si specifica che non può essere avvenuta un'infezione
- se i due elementi i-esimi presentano il primo carattere uguale allora si procede confrontando i due elementi i-esimi (si assuma X vettore relativo alla "sequenza originale" e Y vettore relativo alla sequenze che si vuole dimostrare l'eventuale infezione):
  - se il primo carattere è "a" allora la lunghezza di $Y[i]$ deve essere minore o uguale a 5 volte quella di $X[i]$, in quanto per ogni "a" in $X[i]$ posso avere al più 5 "a" in $Y[i]$
  - se il primo carattere è "t" allora la lunghezza di $Y[i]$ deve essere minore o uguale a 10 volte quella di $X[i]$, in quanto per ogni "t" in $X[i]$ posso avere al più 10 "t" in $Y[i]$
  - se il primo carattere è "c" o "g" allora la lunghezza di $Y[i]$ deve essere maggiore o uguale di quella di $X[i]$, in quanto per ogni "c" o "g" in in $X[i]$ posso avere un numero indefinito di "c" o "g" in $Y[i]$

Si ha quindi il seguente pseudocodice dell'algoritmo, che ritorna $\top$ o $\bot$ a seconda che sia possibile che $seq2$ sia una versione infettata di $seq1$:

Algorithm 2 algoritmo di verifica dell'infezione

```
function CheckInfection(seq1, seq2)
    if length(seq1) == 0 or length(seq2) == 0 then
        return ⊥
    end if
    vecseq1 ← Splitter(lowercase(seq1))
    vecseq2 ← Splitter(lowercase(seq2))
    check ← ⊤
    if length(vecseq1) ≠ length(vecseq2) then
        return ⊥
    end if
    j ← length(vecseq1)
    for i ← 0 to j do
        if vecseq1[i][0] ≠ vecseq2[i][0] then
            return ⊥
        end if
        if vecseq1[i][0] == 'a' then
            check ← (length(vecseq2[i]) ≤ 5 · length(vecseq1[i]))
        else if vecseq1[i][0] == 't' then
            check ← (length(vecseq2[i]) ≤ 10 · length(vecseq1[i]))
        else
            check ← (length(vecseq2[i]) ≥ length(vecseq1[i]))
        end if
    end for
    return check
end function
```

Esempio 2. *Qualche esempio:*

- ***input**: "ATAGCTC" e "AAATAAAGGGGCCCCCTTTTTTTCC"*
- ***output**: $\top$*

infatti (si usano i colori per rappresentare i vari "token" dei vettori):

$$ATAGCTC$$
$$AAATAAAGGGGCCCCCTTTTTTTCC$$

Avendo che tutti i vincoli sono rispettati.

D'altro canto vediamo un esempio in cui non si può dire di avere una mutazione:

- ***input**: "ATAGCTC" e "AAATAAAAAAGGGGCCCCCTTTTTTTCC"*
- ***output**: $\perp$*

infatti (si usano i colori per rappresentare i vari "token" dei vettori):

$$ATAGCTC$$
$$AAATAAAAAAGGGGCCCCCTTTTTTTCC$$

Avendo che il terzo "token" della prima stringa è A mentre quello della seconda stringa è $AAAAAA$, avendo che viene rotto il vincolo per il quale per ogni A posso avere al più 5 A nella mutazione.
Un altro esempio in cui non si può dire di avere una mutazione è:

- ***input**: "ATAGCTC" e "AAACCTAAAAAAGGGGCCCCCTTTTTTT"*
- ***output**: $\perp$*

infatti (si usano i colori per rappresentare i vari "token" dei vettori):

$$ATAGCTC$$
$$AAACCTAAAAAAGGGGCCCCCTTTTTTT$$

In quanto i due secondi "token", T e CC, corrispondono a caratteri diversi.

1.2 Versione 2

Si assumano stringhe su $\Sigma = \{a, c, g, t, A, C, G, T\}$ indicizzate a partire dalla posizione 0, per comodità.
Indico anche una versione intuitivamente più semplice. In questo caso si scorre la sequenze che si suppone essere originale, tenendo conto dei caratteri uguali ripetuti. Appena si ha un cambio di carattere si verifica se nella sequenza che si vuole dimostrare essere una mutazione di quella originale si ha una corretta sequenza di caratteri secondo la specifica.
Si ha che:

- per comodità le sequenze sono riportate in lowercase e si assume che, in presenza di anche solo una sequenza nulla, l'algoritmo restituisce $\perp$
- `seq1` e `seq2` sono rispettivamente la sequenza originale al sequenza che si vuole dimostrare essere la mutazione
- `co` e `cm` sono le variabili che ogni volta accumulano il conteggio dei caratteri uguali consecutivi rispettivamente sulla sequenza originale e su quella che si suppone mutata
- i e j sono rispettivamente gli indici per la prima e per la seconda sequenza

In poche parole si itera sulla sequenza originale, aggiornando di volta in volta il contatore relativo finché si ha lo stesso carattere. Nel momento in cui si ha un cambiamento o si è arrivato all'ultimo carattere della sequenza si ferma il conteggio e si verifica il conteggio sulla seconda sequenza (anche in questo caso facendo attenzione a non andare *out of bounds*). Qualora la seconda sequenza presenti, all'indice a cui si è arrivati con l'algoritmo, un carattere diverso da quello della prima si può restituire $\perp$. Controllando la seconda sequenza si aggiorna il rispettivo contatore e l'indice. Una volta che anche sulla seconda sequenza si ha un cambio di carattere o si è arrivati alla fine si confrontano i due contatori secondo le specifiche (ad esempio se nella stringa originale avevo due "a" consecutive ne posso avere al più 10 in quella che si vuole verificare essere la mutazione). Finito questo controllo si azzerano i contatori e si verifica che, qualora la sequenza originale sia stata visitata interamente, anche la seconda sia conclusa. In caso contrario si restituisce $\perp$.
Si ha quindi il seguente pseudocodice.

Algorithm 3 algoritmo di verifica dell'infezione, seconda versione

```
function CheckInfection(seq1, seq2)
    m ← length(seq1)
    n ← length(seq2)
    if m == 0 or n == 0 then
        return ⊥
    end if
    seq1 ← lowercase(seq1)
    seq2 ← lowercase(seq2)
    co ← 0
    cm ← 0
    j ← 0
    for i ← 0 to m do
        co ← co + 1
        if i == n − 1 or seq1[i] ≠ seq[i + 1 then
            if seq1[i] ≠ seq2[j] then
                return ⊥
            end if
            while j ≠ n − 1 and seq2[j] == seq2[j + 1] do
                cm ← cm + 1
                j ← j + 1
            end while
            cm ← cm + 1
            j ← j + 1
            if seq1[i] == 'a' and cm ≤ 5 · co then
                check ← ⊤
            else if seq1[i] == 't' and cm ≤ 10 · co then
                check ← ⊤
            else if (seq1[i] == 'c' or seq1[i] == 'g') and cm ≥ co then
                check ← ⊤
            else
                return ⊥
            end if
            co ← 0
            cm ← 0
            if i == m − 1 and j ≠ n then
                return ⊥
            end if
        end if
    end for
    return check
end function
```

Dal punto di vista pratico questo secondo algoritmo evita la fase di preprocessing vista nel primo algoritmo con la funzione `Splitter`. Anziché avere a priori le sottostringhe di cui confrontare le lunghezze tiene conto di volta in volta dei caratteri uguali consecutivi, confrontando i contatori (motivo per cui anche gli esempi possono essere riadattati anche a questa seconda versione). Dal punto di vista computazionale, assumendo che la funzione `length` abbia costo lineare si ha in entrambi i casi un tempo quadratico nel caso peggiore, anche se si potrebbe fare uno studio più approfondito.

Capitolo 2

Esercizio 2

Capitolo 3

Esercizio 3

La **distanza di Hamming** tra due stringhe, che si assumono di uguale lunghezza, è il numero di indici per i quali i due caratteri associati sulle due stringhe sono diversi. È quindi un conteggio delle sostituzioni necessarie per passare da una stringa all'altra.
Ipotizzando di voler studiare la distanza di Hamming tra due sequenze tramite la griglia estesa si può ipotizzare di ragionare solo in termini di una delle due sequenze, ovvero considerare i pesi degli archi in ottica di una sola delle due sequenze. Si associano quindi i seguenti pesi:

- $w(diagonale) = 0$
- $w(verticale) = 0$
- $w(orizzontale) = 1$

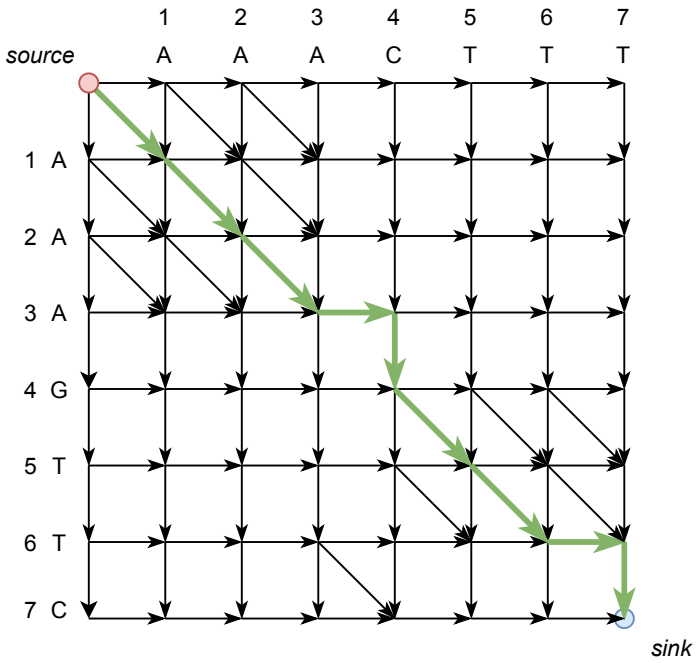
o, in modo speculare se si vuole ragionare sull'altra sequenza:

- $w(diagonale) = 0$
- $w(verticale) = 1$
- $w(orizzontale) = 0$

Dove gli archi diagonali rappresentano un match.
A questo punto si cerca il cammino di peso minimo che parte dal nodo *source*, posto in alto a sinistra, e arriva al nodo *sink*, posto in basso a destra.
Prendendo, ad esempio, in input:

- AAAGTTC
- AAACTTT

Si ha un cammino minimo (non l'unico), assumendo costo degli archi in orizzontale pari a 1 e in verticale pari a 0, del tipo:



Dove in verde è segnato il cammino minimo scelto. Avendo solo due archi orizzontali, che ricordiamo avere peso 1 mentre gli altri hanno peso nullo, possiamo concludere che la distanza di Hamming tra le due stringhe è pari a 2.

Capitolo 4

Esercizio 4

Il problema della **longest common substring** si pone l'obiettivo di estrarre, a partire da due stringhe di lunghezza arbitraria, anche non uguale, la sottostringa, comune ad entrambe, più lunga.
Dal punto di vista della griglia si assegnano i seguenti pesi:

- $w(diagonale) = 1$
- $w(verticale) = -1 \cdot i$
- $w(orizzontale) = -1 \cdot j$

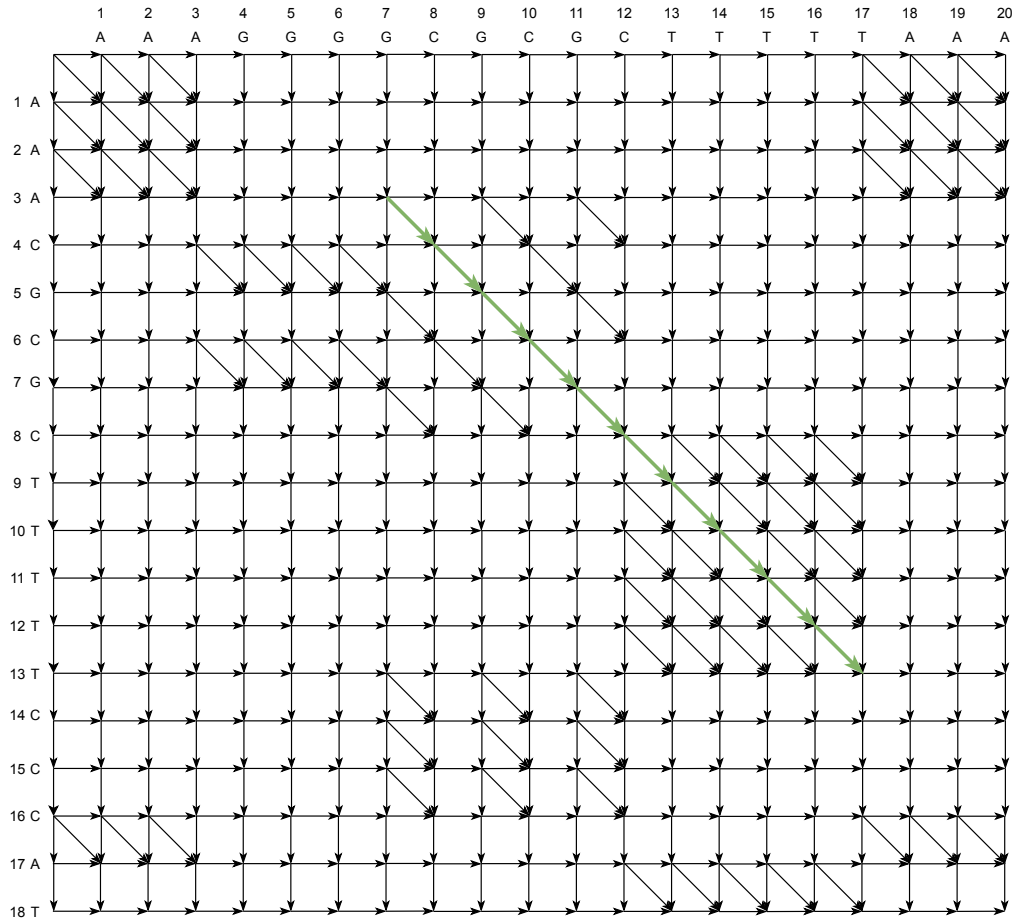
Dove gli archi diagonali rappresentano un match.
Qualora si ottenga un valore negativo viene messo 0.
Si ragiona quindi tenendo traccia del valore massimo raggiungo, tenendo traccia, per una maggior efficienza, anche delle coordinate. Una volta completata la griglia si avrà che il valore massimo corrisponde al nodo *sink* del cammino composto dalla più lunga sequenza possibile di archi diagonali consecutivi. Qualsiasi altra "operazione", con archi orizzontali o diagonali, comporta infatti una perdita di punteggio e ogni volta che un cammino diagonale termina viene azzerato il punteggio, tramite pesi negativi pesati sugli indici. Il valore massimo quindi altro non è che la lunghezza della *longest common substring*. Sapendo che archi diagonali corrispondono a match tra le due sequenze si ottiene che tale cammino corrisponde alla *longest common substring*.
Volendo si può scegliere di salvare in un vettore i valori massimi qualora coincidano, per ottenere eventuali più *longest common substring* qualora ce ne siano.
Per ricostruire la *longest common substring* si parte dal nodo *sink*, come detto definito dal valore massimo calcolato. Si aggiunge il carattere corrispondente e si risale tutto il cammino diagonale, aggiungendo di volta in volta in testa

il carattere letto. Ci si ferma quindi quando non si ha più un nodo il cui punteggio è stato ottenuto tramite un arco diagonale.

Vediamo quindi un esempio pratico. Siano date:

- AAACGCGCTTTTTCCCAT
- AAAGGGGCGCGCTTTTTAAA

Si costruisce quindi la griglia e si identifica il cammino diagonale più lungo:



Ricostruendo si ha che tale cammino identifica un massimo pari a 10 e si ricostruisce:

CGCGCTTTTT

che è appunto la *longest common substring* delle due sequenze in input.