

# Relazione Progetto

## Elementi di Bioinformatica

### Long Bitap

Davide Cozzi  
829827  
d.cozzi@campus.unimib.it

## Bitap

*Nell'analisi un bit rappresentato in colonna presenta il bit più significativo, MSB, in basso*

Innanzitutto si descrive l'algoritmo di base, funzionante per pattern lunghi al massimo quanto una *word*,  $w$  della cpu.

Si hanno in input un *pattern*  $P$  di lunghezza  $p$  e un *testo*  $T$  di lunghezza  $t$ . Si ha quindi  $p \leq w$ .

A livello teorico si costruisce una **matrice booleana**  $D$ , di dimensioni  $p \times t$ , e si stabiliscono due indici:

1.  $i$  che itera sul pattern
2.  $j$  che itera sul testo

Si ha che la generica posizione di indici  $i, j$  nella matrice è 1 sse i primi  $i$  caratteri del pattern matchano un numero  $i$  di caratteri del testo terminanti all'indice  $j$ .

Si ha quindi la seguente **equazione di ricorrenza**:

$$D[i, j] = \begin{cases} 1 & \text{sse } P[1..i] = T[j - i + 1..j] \\ 0 & \text{altrimenti} \end{cases}$$

Nell' $i$ -sima riga si ha che le occorrenze di 1 indicano i punti nel testo dove termina una copia di  $P[1..i]$ .

Invece la  $j$ -sima colonna mostra tutti i prefissi del testo che finiscono nella posizione  $j$  del testo.

Nell'ultima riga della matrice si ha la soluzione, ovvero si ha 1 dove termina un match del pattern nel testo.

In termini pratici questo algoritmo può essere costruito mediante operazioni **bit a bit** in quanto le singole colonne della matrice teorica possono essere viste come numeri in rappresentazione binaria.

Per procedere si ha una fase di preprocessamento in cui si costruisce un array  $U$ , di lunghezza pari a quella dell'alfabeto in uso, che contiene in posizione  $k$  il binario rappresentante le occorrenze del carattere  $k$  nel pattern. Nel pattern il vettore  $U$  viene definito di lunghezza `CHAR_MAX`, ovvero 127, per usare la tabella ASCII a 7 bit. Per costruire tale array si procede inizializzando tutte le celle a 0. Si itera poi lungo il pattern aggiornando  $U$  nella posizione del carattere preso in considerazione del pattern facendo l'**or** tra l'attuale contenuto di  $U$  in quella posizione e il numero la cui rappresentazione binaria presenta 1 solo nella posizione di indice  $i$  (questo comportamento è ottenibile con il *left-shift* di 1 di *posizioni*). Dopo aver iterato su tutto il pattern ottengo il vettore  $U$  correttamente caricato.

**Esempio 1.** *Immaginiamo un pattern semplice: caac.*

*Il primo carattere è c e i è in posizione 0. U è ancora caricato con soli zeri. Quindi si ha:*

$$U[c] = U[99] = \begin{array}{ccc} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array} \vee \begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array} = \begin{array}{ccc} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}$$

*proseguo con a e shift i di 1:*

$$U[a] = U[97] = \begin{array}{ccc} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{array} \vee \begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array} = \begin{array}{ccc} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{array}$$

*il carattere successivo è ancora a:*

$$U[a] = U[97] = \begin{array}{ccc} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{array} \vee \begin{array}{ccc} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{array} = \begin{array}{ccc} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{array}$$

*e infine trovo ancora c:*

$$U[c] = U[99] = \begin{array}{ccc} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array} \vee \begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{array} = \begin{array}{ccc} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{array}$$

Alla fine quindi  $U[c] = 1001|_2 = 9$  e  $U[a] = 0110|_2 = 6$

L'algoritmo prosegue inizializzando la prima colonna a 1 se testo e pattern condividono il primo carattere, 0 altrimenti.

Si procede poi col calcolo delle colonne successive alla prima sfruttando la colonna precedente. Si procede con un *left-shift* del valore rappresentante la colonna precedente con l'aggiunta di un 1 in testa. Si procede poi con l'**and** tra il risultato appena ottenuto e il valore di U nella posizione del carattere che sto considerando. Si controlla infine ogni valore rappresentante una colonna vedendo se presenta 1 nell'ultimo bit. Per ottenere questo risultato si procede con il *left-shift* di uno di un'unità pari alla lunghezza del pattern meno uno e all'**and** con il valore rappresentante la colonna. Mi verrà infatti restituito un binario avente valore 0 o  $2^{p-1}$  (nel caso di word grandi 3 avrei  $100|_2 = 4$  avendo quindi 1 nell'ipotetica ultima riga della matrice, avendo quindi un match).

Il limite di questo algoritmo è hardware e consiste nella rappresentazione (e quindi anche nelle operazioni) su binari oltre il numero di bit della word.

Con questa soluzione si ha un tempo di  $\sim O(p + t)$

## Slow Long Bitap

*Questa è la mia implementazione più naive per superare il limite della grandezza della word.*

Procedo innanzitutto separando il pattern in *sottopatterns* lunghi  $w$ , tranne l'ultimo, che sarà lungo  $p - w$ . Spezzo quindi il pattern in un numero di sottopatterns pari a  $\lceil \frac{p}{w} \rceil$ . Questi patterns vengono caricati in un array di stringhe.

Per effettuare il match si sfruttano 3 array di lunghezza  $t$ . Si ha un array contenente l'ultima riga teorica prodotta dall'algoritmo **bitap**, uno contenente quello precedente e uno risultante. Si procede a coppie verificando che l'array dell'ultimo bitap presenti 1 esattamente ad una distanza pari alla lunghezza del sottopattern in analisi rispetto ad un 1 nell'array prodotto dall'algoritmo **bitap** sul sottopattern precedente. Nel caso si abbia questa corrispondenza tra i due array si carica 1 nel terzo array (quello risultante) in corrispondenza dell'indice in cui c'era 1 nell'array prodotto per il sottopattern corrente, altrimenti si carica 0. Alla successiva iterazione l'array risultante diventerà quello precedente fino ad esaurimento dei sottopatterns. A questo punto avrò un array risultante con gli indici che rappresentano la fine di un match di  $P$  in  $T$ . Per ottenere le posizioni di inizio basta sottrarre a tali valori  $p - 1$  (il  $-1$  è causato dall'inizio all'indice 0 e non 1).

**Esempio 2.** Vediamo un esempio semplificato ipotizzando una cpu con  $WORDSIZE = 2$ .

Sia  $T = \text{abbaccabbacabcabbacc}$  e  $P = \text{abbac}$ .

Si avrà quindi il seguente array dei sottopatterns:

$$\text{patterns} = \{ "ab", "ba", "c" \}$$

Per semplicità rappresentiamo la tabella complessiva dei tre **bitap** senza rappresentare i vari step intermedi. Si otterrebbe quindi la matrice dei 3 bitap (dove vengono sottolineate le occorrenze di uno valide secondo la logica sopra descritta):

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	a	b	b	a	c	c	a	b	b	a	c	a	b	c	a	b	b	a	c	c
<i>ab</i>	0	<u>1</u>	0	0	0	0	0	1	0	0	0	0	1	0	0	<u>1</u>	0	0	0	0
<i>ba</i>	0	0	0	<u>1</u>	0	0	0	0	0	1	0	0	0	0	0	0	0	<u>1</u>	0	0
<i>c</i>	0	0	0	0	<u>1</u>	1	0	0	0	0	1	0	0	1	0	0	0	0	<u>1</u>	1

Si ha quindi un match terminante all'indice 4 (quindi iniziante all'indice 0) e uno all'indice 18 (quindi iniziante all'indice 14)

Visto che viene effettuata una chiamata a **bitap** per ogni sottopattern si ha un tempo pari a  $\sim O\left(\lceil \frac{p}{w} \rceil \cdot (p + t)\right)$ .

Si notano quindi le problematiche che posso nascere all'aumentare della grandezza del testo e a quella del pattern,