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 <= w`.
A livello teorico si costruisce una **matrice booleana** `D`, di dimensioni $\texttt{p} \times \texttt{t}$, 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**:

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

Nell'`i`-sima riga si ha che le occorrenze di 1 indicano i punti nel testo dove termina una copia di $\texttt{P}[1..\texttt{i}]$.
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:*

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

proseguo con `a` *e shifto* `i` *di 1:*

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

il carattere successivo è ancora `a`*:*

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

e infine trovo ancora `c`*:*

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

Alla fine quindi $U[\texttt{c}] = 1001|_2 = 9$ *e* $U[\texttt{a}] = 0110|_2 = 6$

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 o 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, infatti viene fatto un `and` con un valore che in codifica binaria presenta tutti 0 tranne il *MSB* che presenta 1).
Il limite di queto 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 $\left\lceil \frac{\texttt{p}}{\texttt{w}} \right\rceil$. 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 $\mathtt{p} - 1$ (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 $\mathtt{T} = abbaccabbacabcabbacc$ *e* $\mathtt{P} = abbac$.
Si avrà quindi il seguente array dei sottopatterns:

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

Per semplicità rappresentiamo la tabella complessiva dei tre **`bitap`** *senza rappresentare i vari steps 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*
ab	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*1*	*0*	*0*	*0*	*0*
ba	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0*
c	*0*	*0*	*0*	*0*	*1*	*1*	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*1*	*1*

Si ha quindi un match terminante all'indice 4 (quindi iniziante all'indice 0), uno all'indice 10 (quindi iniziante all'indice 6) 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 (ipotizzando un tempo $O(m)$, con $m \sim t$, per la `memcpy`):

$$\sim O\left(\left\lceil \frac{\mathtt{p}}{\mathtt{w}} \right\rceil \cdot (m + w + 2 \cdot t)\right) \rightarrow \sim O\left(\left\lceil \frac{\mathtt{p}}{\mathtt{w}} \right\rceil \cdot (m + w + t)\right)$$

Si notano quindi le problematiche che posso nascere all'aumentare della grandezza del testo e a quella del pattern.
Il problema grave di questa implementazione è che vengono effettuate troppe operazioni inutili a priori chiamando bitap su tutto il testo per tutti i sottopattern.
In termini pratici (ma meno oggettivi) sulla mia macchina (*Intel i7 8550U* con 8gb di ram) per un pattern di 3374 caratteri su un testo lungo 3391270 si ha un tempo di circa 2.2 secondi per trovare le 60 occorrenze.

Fast Long Bitap

Il ragionamento di base è lo stesso della precedente implementazione e si procede spezzando il pattern in maniera analoga. La differenza sostanziale si ritrova nella finestra di testo su cui viene chiamata la funzione `bitap` e il numero di volte in cui essa viene chiamata.
Innanzitutto viene chiamata la funzione `bitap` sul primo sottopattern (qualora si abbia un solo sottopattern si è nella situazione in cui `p <= w` e quindi si stampano direttamente le occorrenze). Si chiama anche una funzione `countfirst` che, preso in ingresso un array, restituisce un secondo array contenente gli indici dove si ha un 1. Chiamando questa funzione sull'array restituito dalla funzione `bitap`, chiamata sul primo sottopattern, calcolo gli indici in cui termina un match del primo sottopattern nel testo. Questo passaggio è **fondamentale** per ridurre il numero di operazioni inutili in quanto a priori gli eventuali match dell'intero pattern avranno come indice di partenza gli indici ricavabili come sopra a partire da questi ultimi.
Procedendo col ragionamento si ha che si può ricercare il secondo sottopattern solo nella finestra di testo, che parte con l'indice di fine match del primo sottopattern, di lunghezza pari a questo secondo sottopattern. A questo punto chiamo quindi `bitap` su questa piccola finestra di testo e ne valuto solo l'ultimo elemento. Si hanno quindi 3 casi:

1. l'ultimo elemento è 1 e ho un altro sottopattern da analizzare. Proseguo quindi iterativamente fino ad esaurimento dei sottopatterns

2. l'ultimo elemento è 1 e non ho un altro sottopattern da analizzare. In tal caso ho analizzato l'intero pattern e nell'ultimo indice raggiunto termina un match del pattern sul testo, che viene stampato. A questo punto vedo se esiste un ulteriore valore nell'array contenente gli indice di fine match del primo sottopattern. Se esiste ricomincio da capo a partire da quell'indice altrimenti termino l'esecuzione in quanto ho concluso l'analisi

3. l'ultimo elemento è 0. In tal caso a priori smetto di cercare un pattern iniziante da un dato valore dell'array contenente gli indice di fine match del primo sottopattern ed eventualmente mi sposto al valore successivo ricominciando da capo la ricerca a partire dal nuovo indice. Qualora tale valore non esista termino l'esecuzione in quanto ho concluso l'analisi

Si nota che le chiamate a `bitap` vengono effettuate su un pattern e un testo di uguale lunghezza.

Esempio 3. *Vediamo un esempio semplificato ipotizzando una cpu con* $WORDSIZE = 2$.
Sia `T` $=$ *abbaccabbacabcabbacc e* `P` $=$ *abbac.*
Si avrà quindi il seguente array dei sottopatterns:

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

Chiamo **`bitap`** *sul primo sottopattern, ovvero "ab":*

	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
ab	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0

l'array calcolato con **`countfirst`** *conterrà i valori:*

$$\{1, 7, 12, 15\}$$

Parto quindi dall'indice 1. Chiamo bitap sul secondo sottopattern, "ab", lungo 2, nella porzione di testo tra 2 e 3 (inclusi). La chiamata **`bitap`** *di "ba" su "ba" restituisce* $\{0, 1\}$. *Avendo 1 in ultima porzione posso proseguire. Chiamo* **`bitap`** *il terzo sottopattern, "c", lungo 1, sulla finestra di testo che parte da 4 e termina in 4 (inclusi). Ovviamente essendo "c" uguale a "c" ottengo l'array con solo un elemento:* $\{1\}$. *Avendo finito i sottopattern e avendo 1 alla fine so che ho un match terminante in posizione 4 (e quindi iniziante in posizione 0).*
Mi sposto quindi sul secondo indice di match del primo sottopattern: 7. Procedendo come sopra scopro che anche in questo caso ho un match, terminante all'indice 10 (e quindi che inizia all'indice 6).
Valuto quindi 12. Chiamo bitap sul secondo sottopattern, "ab", lungo 2, nella porzione di testo tra 2 e 3 (inclusi). La chiamata **`bitap`** *di "ba" su "ca" restituisce* $\{0, 0\}$. *Smetto quindi di analizzare altro e passo a valutare 15. Anche in questo caso si avrà un match, terminante in 18 e iniziante in 14.*
Ho quindi trovato gli indici di partenza dei 3 match:

$$\{0, 6, 14\}$$

Diamo ora una stima approssimata dei tempi. Chiamo k il numero di occorrenze del primo pattern nel testo. Sapendo che il costo del calcolo di una sottostringa ha un costo pari alla lunghezza della sottostringa (e sapendo che in generale queste saranno lunghe `w`) si ha che ogni chiamata alla funzione che calcola la sottostringa costa $\sim O(\texttt{w})$. Come detto la funzione `bitap` viene chiamata su un testo lungo quanto il sottopattern (che sappiamo essere lungo

al più `w`), quindi il suo costo è $\sim O(2 \cdot w) \rightarrow\sim O(w)$. Quindi il costo dell'algortimo è di (imponendo che n rappresenta il numero di sottopatterns, ovvero $n = \left\lceil \frac{\mathtt{p}}{\mathtt{w}} \right\rceil$):

$$\sim O(k \cdot n \cdot (2w)) \rightarrow\sim O(k \cdot n \cdot w)$$

a questa stima vanno sommati i costi della prima chiamata a `bitap`, ovvero $\sim O(w + t)$ e del calcolo deglin indici di partenza, ovvero $\sim O(t)$. Si nota come non si abbia più `t` moltiplicato per il numero di sottopatterns (fattore che rallentava l'algoritmo in presenza di testi molto lunghi).

Tabella Test

In termini pratici (ma meno oggettivi) sulla mia macchina (*Intel i7 8550U* con 8gb di ram) sono stati effettuati 4 test:

1. **short**: un breve pattern (di grandezza minore a word) su un breve testo:
    - lunghezza pattern: 7
    - lunghezza testo: 26
    - numero di occorrenze: 2
2. **word**: un breve pattern lungo quanto una word su un testo non breve:
    - lunghezza pattern: 64
    - lunghezza testo: 74640
    - numero di occorrenze: 289
3. **medium**: un pattern lungo circa il triplo di una word su un testo non breve:
    - lunghezza pattern: 151
    - lunghezza testo: 37287
    - numero di occorrenze: 170
4. **long**: un pattern lungo su un testo molto lungo:
    - lunghezza pattern: 3374
    - lunghezza testo: 3391270
    - numero di occorrenze: 60

Vediamo quindi una tabella riassuntiva coi tempi calcolati da `time`:

	short	word	medium	long
slow	$\sim 0.001s$	$\sim 0.004s$	$\sim 0.005s$	$\sim 1.6s$
fast	$\sim 0.001s$	$\sim 0.004s$	$\sim 0.004s$	$\sim 0.06s$

Questi dati non sono universalmente utili ma sono identificativi del miglioramento delle performance in presenza di pattern e testo lunghi