

Elementi di Bioinformatica



Lezione 1 - Bit parallel

Notazione

Simbolo $T[i]$

Stringa $T[1] \dots T[\ell]$

Sottostringa $T[i:j]$

Prefisso $T[:j]$

Suffisso $T[i:]$

Concatenazione $T_1 T_2$

Problema: Pattern Matching

Input: Ho un testo $T = T[1], \dots, T[n]$ ed un pattern $P = P[1], \dots, P[m]$

Goal: Trovare tutte le occorrenze P in T

Lower Bound: Il minor tempo di calcolo possibile è $\Theta(n+m)$ perché devo almeno leggere T e P

Con un algoritmo banale (uso dei for) otterrei $\Theta(n \cdot m)$. Introduciamo un algoritmo che seppur con tempo $\Theta(n \cdot m)$, è nell'effettivo più veloce in quanto vengono eseguiti pochi calcoli dalla cpu. Costante moltiplicativa minore.

Prima di introdurre l'algoritmo, introduciamo una forma di calcolo definita bit-parallel: eseguiamo 3 operazioni parallelamente su tutti i bit della macchina. Utilizzeremo come operazioni:

- and: $x \wedge y$
- or: $x \vee y$
- xor: $x \oplus y$
- right-shift: $x \gg k$
- left-shift: $k \ll x$

Lezione 1 - Bit parallel

Algoritmo di DomoK:

- Costruiamo una matrice con sulle righe i prefissi P e sulle colonne i caratteri del testo T . Inserisco \sqcup in posizione $M(i,j)$ sse $P[:i] = T[j-i+1:i]$ avendo se la lunghezza del testo fissato il pattern termina con lo stesso carattere:

	A	B	R	A	C	A	D	A	B	R	A
A	$\sqcup A$	O	O	$\sqcup A$	O	$\sqcup A$	O	$\sqcup A$	O	O	$\sqcup A$
AB	O	$\sqcup AB$	O	O	O	O	O	O	$\sqcup AB$	O	O
ABR	O	O	$\sqcup ABR$	O	O	O	O	O	O	$\sqcup ABR$	O

- Nota che per controllare se $M(i+1, j+1)$ matcha, devo solo considerare l'ultimo carattere, in quanto i caratteri precedenti sono stati già controllati da $M(i, j)$.

	A	B	R	A	C	A	D	A	B	R	A
A	$\sqcup A$	O	O	$\sqcup A$	O	$\sqcup A$	O	$\sqcup A$	O	O	$\sqcup A$
AB	O	$\sqcup B$	O	O	O	O	O	O	$\sqcup B$	O	O
ABR	O	O	$\sqcup B$	O	O	O	O	O	O	$\sqcup B$	O

- L'algoritmo che definisce ciò, effettua le seguenti operazioni:

3.1 Right-shift di $C[j-s]$

3.2 Inserisco \sqcup in cima

3.3 Eseguo AND tra $C[j]$ e il valore del right-shift

	A	B			A	B
A	\sqcup	$\sqcup \wedge B=A$			A	\sqcup
AB	O	$\sqcup \wedge B=B$	→	AB	O	\sqcup
ABR	O	$\sqcup \wedge B=R$		ABR	O	O

- Ottimizzo ulteriormente salvandomi le codifiche dell'alfabeto del testo: $U[\sigma]$

- Ottengo in forma compatta la seguente istruzione:

$$C[j] = ((C[j-s] >> s) | (s << C[j-s])) \wedge U(T(j))$$

Lezione 2 - Karp-Rabin

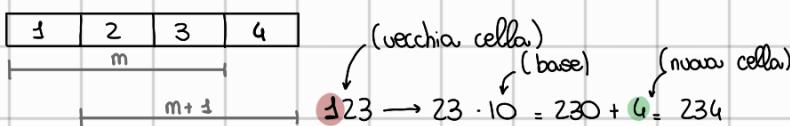
Dopo aver visto DomoKij, affrontiamo un nuovo algoritmo dove accetto come caso pessimo $O(n \cdot m)$ ma sarà raro finirci.

Introduzione all'idea Karp-Rabin: Data una stringa su alfabeto binario (per semplicità), considero la sua fingerprint, ossia il suo valore numerico, e uso una sliding window di ampiezza m su T :

$$\cdot H(S) = \sum_{i=1}^{|S|} 2^{i-1} H(S[i])$$

$$\cdot H(T[i:i+m]) = (H(T[i:i+m-1] \cdot T[i]) / 2 + 2^{m-1} T[i+m])$$

Ovviamente, la sliding window successiva controlla solo l'ultima posizione, in quanto conosce il valore delle posizioni precedenti:



Effettuando solo operazioni algebriche su bit mi assicuro che è una implementazione valida, ma purtroppo non efficiente in quanto con m molto grande il costo delle operazioni non sarebbe unitario ma logaritmico, in quanto proporzionale al numero di bit.

Per ottenere tempi costanti faccio tutte le operazioni mod p primo casuale, ottenendo tutti i numeri $< p$:

· **problema**: ci sono più numeri che mod p sono uguali ($6 \equiv 1 \pmod{5}$ e $11 \equiv 1 \pmod{5}$) → genero falsi positivi.

· **soluzione**: utilizzo K numeri primi: così facendo ho il tempo moltiplicato per K , ma il tasso di errore diminuisce a q^K . Questo è possibile dato che siamo sicuri che non esistano falsi negativi.

Cambio p ogni volta che p genera FP. Se la sequenza genera FP per tutti i K p , allora è FP.

Osservazione: I migliori primi p da scegliere casualmente sono i più grandi vicini alla word size perché generano meno valori uguali.

Classificazione algoritmi probabilistici

· MonteCarlo: sempre veloce, non sempre corretto. Es. Karp-Rabin

· Las Vegas: non sempre veloce, sempre corretto Es. Quicksort con Pivot random.

Oss: Posso passare da MC a LV aggiungendo controlli, non vale viceversa.

Lezione 3 - Suffix-Tree-Array

Dopo aver visto due algoritmi con caso peggiore $O(n \cdot m)$, con il nuovo algoritmo riusciremo ad arrivare al lower bound $O(n+m)$. Cioè nonostante, si dimostrerà che nella pratica non è poi così veloce, e bisognerà "ribassare" la teoria per migliorarlo.

Introduciamo i tries: albero dove gli archi sono etichettati con le lettere, la query sull'albero consiste nel verificare se una parola appartiene all'albero, cioè se esiste il cammino radice-foglia.

problema: non riesco ad identificare le prefissi, in quanto non termino su una foglia.

Soluzione: aggiungo \$ alla fine delle parole, dove \$ è Vocabolario

Il trie è ottimo in quanto il tempo della query non cambia, dipende solo dal pattern, indipendente dal vocabolario

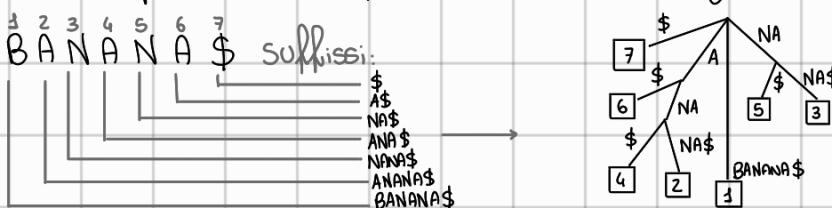
Il legame tra trie e pattern match è che ogni sottostringo è prefisso del suffisso. Se riesco a costruire il trie di tutti i suffissi risolvo il problema: tra i suffissi, i suffissi tali che il prefisso lungo m è uguale al pattern:

se lo costruisco su un generico dizionario, non ho legami

se lo costruisco sui suffissi, questi sono legati tra loro

Suffix tree: trie compatto di tutti i suffissi $T\$$: le etichette degli archi uscenti da x hanno iniziali diverse.

Esempio: B A N A N A \\$ suffissi:



Osservazione: affinché il pattern faccia match non è necessario consumare tutte le lettere dell'arco:

se il pattern è NAN match con NANA senza consumare la A.

Definizioni utile:

- path-label(x): concatenazione etichette
- string-depth(x): lunghezza path-label(x)
- pattern-matching(x): visita

Lezione 3: Suffix-Tree - Array

Problema: $O(n^2)$ di spazio richiesto

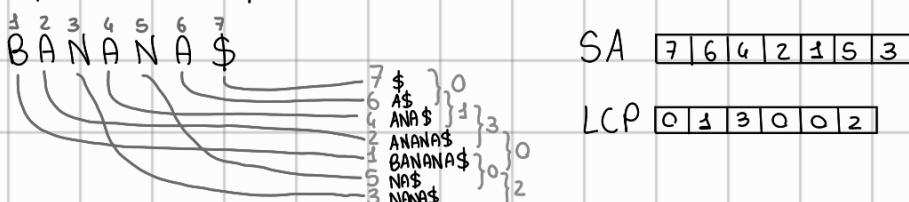
Soluzione: Solviamo gli archi con gli indici.

utilizza 3 puntatori per inizio e uno per length

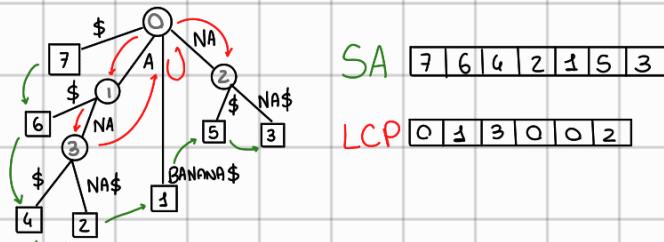
Problema: spazio per carattere 20 n bytes \rightarrow genoma umano in 128 gb e brutta località

Soluzione: suffix-array, bastano 4n bytes \rightarrow genoma umano in 36 gb e buona località

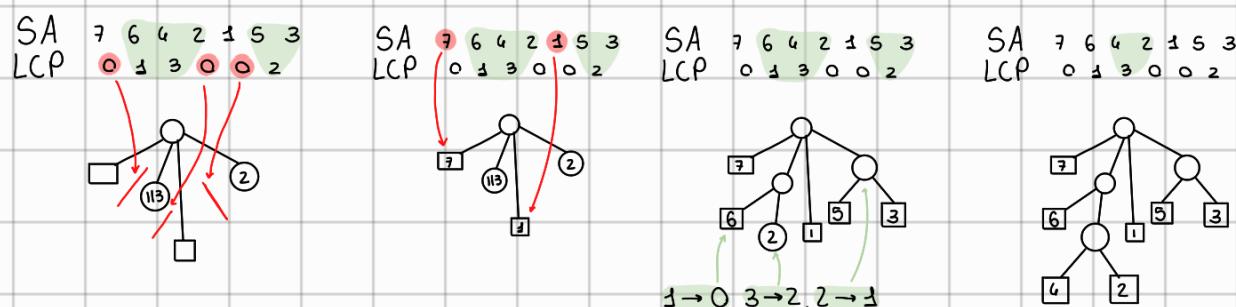
Suffix-array: Array dei suffissi in ordine lessicografico, posizioni iniziali del suffisso memorizzati. Inoltre salvo con $Lcp[i]$: lunghezza prefisso comune tra due suffissi consecutivi $SA[i], SA[i+1]$



Suffix tree \rightarrow Suffix Array: Per costruire il SA partendo da ST mi basta visitare in profondità e ordire l'albero. Per LCP ogni volta che salgo/scendo lungo un arco, tolgo/aggiungo la string-depth. LCP serve per poter attraversare in modo efficiente sia top-down che bottom-up

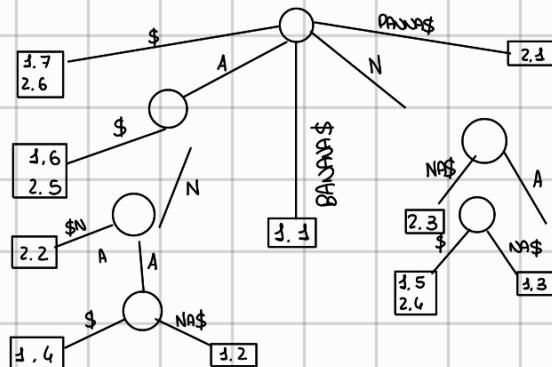


Suffix array (+ LCP) \rightarrow Suffix-tree: Prendiamo LCP: ogni volta che tra 0 so che ho una nuova partizione (prefissi diversi). Altrimenti gli elementi di LCP indicano il numero di nodi mancanti, a cui sottraggo 1. Da eseguire ricorsivamente



Lezione 3 - Suffix-Tree-Array

Suffix-tree generalizzato. Posso prendere ora più stringhe e generalizzare il problema LCS sul suffix-tree generalizzato. Esempio con BANANA\$ e PANNA\$



Longest Common Subsequence su Suffix-Tree generalizzato: (non pattern matching!)

Solo nelle foglie ho parole che considero (z, z, \dots, n) e le posizioni $(1, \dots, m)$. Come si fa LCS?

Le concateno e distinguo i dollari: BANANA\$, PANNA\$₂ e genero SF. Ora posso avere due tipi di sottostringhe: quelle che sono suffisso solo a w₂ oppure a entrambe: BANANA\$, PANNA\$₂

Se un nodo è suffisso di una stringa, lo è anche il nodo padre: etichetto ogni nodo con le stringhe della quale è suffisso in bottom-up (ottimizzo): metto T/F.

Per effettuare LCS leggo l'albero e cerco nodo con le etichette delle stringhe e string-depth maggiore (top-down)

PM SA vs PM ST: tree $O(m+k)$ vs array $O(m \log n + k)$ con $m = |\text{pattern}|$ e $n = |\text{text}|$.

- tree percorre l'albero (lineare)
- array ricerca dicotomica (si aggiunge log)

Riassunto idea PM su GST: solo in ogni nodo un array di K booleani dove K è il numero di parole. Se array[i]=1, allora il nodo ha almeno una foglia che contiene la i-esima parola

Problema: per ogni nodo abbiamo $K \cdot n$ operazioni e quindi tempo $O(K \cdot n^2)$?

Soluzione: ricordando che ogni nodo ha un solo padre e quindi ogni nodo viene letto 1 volta come padre e 1 come figlio. $O(n \cdot K^2)$ in visita bottom-up.

Lezione 4 - P1 - Suffix-Array

Ideas: Per fare P1 su SA posso fare ricerca dicotomica $O(\log_2 n)$ e dovrei controllare tutto $O(m)$ ottenendo tempo $O(m \log_2 n)$. Vogliamo velocizzare.

Acceleranti: Introduciamo 3 acceleranti per ridurre il tempo (i primi due nell'effettivo, l'ultimo a livello teorico)

Accelerante 1: Avendo un SA e un suo intervallo $SA(L, R)$ di elemento mediano M. Se $S[L]$ e $S[R]$ cominciano con n caratteri, allora i primi n caratteri del loro intervallo sono uguali e posso evitare di confrontarli.

Problema: non ho niente nel LCP che mi dice il numero di caratteri uguali tra L e R.

Accelerante 2: Denoto $l = lcp(L, P)$ e $r = (R, P)$ dove P è il pattern. Mi posso trovare in 3 casi:

Caso $l > r$: Calcolo $m = lcp(L, M)$

1. $l > m$: faccio match sopra m e quindi $r = m \wedge R = M$

2. $l < m$: faccio match + lungo sotto m e quindi $L = M$

3. $l = m$: confronto dal carattere $l+1$

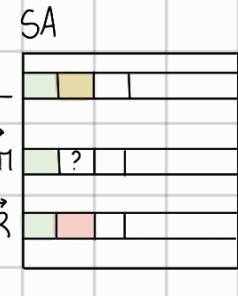
Caso $l < r$: simmetrica a $l > r$

Caso $l = r$: calcolo $m = lcp(L, M)$ e $m' = lcp(M, R)$

$l < m$: faccio match + lungo sotto a M e quindi $L = M$

$l < m'$: caso opposto al precedente: $R = M$

$m = m'$: confronto il carattere successivo



Teoria: ricerca logaritmica e numero costante di operazioni, impiego $O(\log n)$?

Pratica: abbiamo aggiornare l e r 2m volte al massimo. Impiego di più di $O(\log n)$

Problema: come faccio ad ottenere i valori necessari per aggiornare l e r?

Soluzione: se riesco a pre processare il SA per ottenere $lcp(L, M)$ ci impiego $O(m + \log n)$

aggiorno ricerca
 \downarrow \downarrow

Lezione 4 - P1 - Suffix-Array

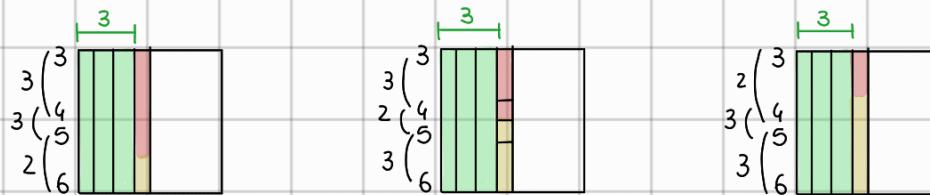
Accelerante 3: Devo considerare il mio array sempre /2.

L
1)
2)
3)
4)
5)
6)
7)
R
8)

Quando calcolo $LCP(L, R)$ ho due casi:

- se L, R sono adiacenti uso LCP

- altrimenti: considero di saper risolvere i problemi + piccoli (op). se so $LCP(1, 4)$ e $LCP(5, 8)$ voglio calcolare $LCP(1, 8)$ posso usare i casi adiacenti intermedi. nell'esempio $LCP(4, 5)$. Posso trovarmi in 3 casi:



Posso confermare che $LCP(5, 8) = \min\{LCP(3, 4), LCP(4, 5), LCP(5, 8)\}$

Conclusioni: Posso processare in tempo lineare $O(n)$ e effettuare la richiesta richiede $O(m \log n)$

Sottostringa comune più lunga. Abbiamo visto come trasformare il PM dal GST al SA per migliorare spazio e località. Ora poniamo la stessa trasformazione per il LCS.