Algoritmi per stringhe

Gabriele Farina
gabr.farina@gmail.com

29 maggio 2021

Indice

1 Hashing

In generale, l'idea dell'hashing (*macinazione*) è quella di "digerire" un oggetto, producendone un numero intero (piccolo), in modo deterministico. Dal momento che stiamo mappando un oggetto grande in un numero piccolo, stiamo potenzialmente perdendo informazioni; d'altra parte maneggiare numeri piccoli (32 o 64 bit tipicamente) è un compito molto facile per un computer.

Dati due oggetti A e B, e i rispettivi hash $h(A)$ e $h(B)$, si pongono due casi:

- se $h(A) \neq h(B)$, sicuramente $A \neq B$;
- se $h(A) = h(B)$, non possiamo concludere nulla. D'altra parte, se la funzione di hash scelta è sensata, sembra ragionevole avere più fiducia nel fatto che $A = B$.

1.1 Monte Carlo vs Las Vegas

Se gli hash coincidono, il test non è conclusivo, ed è necessario decidere come procedere.

Approccio **Las Vegas**: ogni volta che gli hash coincidono, controlliamo se gli oggetti sono uguali, utilizzando un operatore di uguaglianza esatto. *Esempio: per le stringhe, confrontiamo carattere per carattere.*

Approccio **Monte Carlo**: ogni volta che gli hash coincidono, ripetiamo il test con altri hash diversi. Se tutti coincidono, ci arrendiamo all'evidenza che molto, molto probabilmente gli oggetti sono uguali. Ad ogni modo, non abbiamo davvero una garanzia.

1.2 Rolling hash (Rabin-Karp)

Una funzione di hash molto usata per le stringhe è la funzione di hash di Rabin-Karp. Data una stringa

$$s = s_1 \; s_2 \; \cdots \; s_n$$

il suo hash è calcolato in questo modo:

$$h(s) = (s_1 \, R^{n-1} + s_2 \, R^{n-2} + \cdots + s_n) \mod Q$$

dove R è un intero e Q è un numero primo[1]. Questo tipo di hash è detto *rolling* perché si presta bene ad essere calcolato in una finestra scorrevole. In particolare:

[1] D'ora in poi daremo per scontato che tutti i conti saranno da intendersi modulo Q.

- quando aggiungiamo un carattere s_{n+1} a destra della stringa, il nuovo hash si calcola in tempo costante come

$$h(s') = h(s)\, R + s_{n+1};$$

- quando rimuoviamo il carattere s_1 più a sinsitra di una stringa di lunghezza n, il nuovo hash si calcola come

$$h(s') = h(s) - s_1\, R^{n-1}.$$

Alla luce di questo, è facile capire come iterare in ordine su tutte le sottostringhe di lunghezza n (fissata) di una stringa, passando da una alla successiva in tempo costante. Questa proprietà giustifica l'uso dell'hash di Rabin-Karp nell'ambito dell'omonimo algoritmo di string matching.

In termini più astratti, possiamo pensare costruire un oggetto `RKHasher` che supporta l'interfaccia di una *deque*.

1.3 ♣ Uniq

> Date N stringhe in input, dividerle in classi di equivalenza secondo il normale operatore di uguaglianza.

Calcoliamo l'hash di tutte le parole $h(W_1), \ldots, h(W_N)$. A questo punto il problema è equivalente (almeno secondo un approccio Monte Carlo) a suddividere N interi in classi di equivalenza secondo il normale operatore di uguaglianza tra interi. Questo si fa facilmente riordinando i numeri.

1.4 ♣ Matching

> Dato un pattern s e un testo t, listare tutte le occorrenze di s in t.

Usando `RKHasher`, possiamo iterare su tutte le finestre di lunghezza $|s|$ presenti in t. Ogni volta che l'hash della finestra coincide con l'hash di s precalcolato, (molto probabilmente) abbiamo individuato un'occorrenza. Questo porta ad un semplice algoritmo di complessità $\mathcal{O}(|s| + |t|)$, cioè lineare nella dimensione dell'input, e overhead di memoria costante.

1.5 ♣ Ricerca di palindromi

> Data una stringa s, determinare il numero di sottostringhe palindrome di s.

Data una stringa s, determinare una sottostringa palindroma di lunghezza massima.

La soluzione dei due problemi è lasciata come esercizio[2]. È possibile determinare due algoritmi molto simili che, usando la tecnica dell'hashing, risolvono i due problemi in tempo $\mathcal{O}(|s| \log |s|)$.

2 Dictionary of Basic Factors (DBF)

Il DBF di una stringa s è una struttura dati semplice e potente, che permette di confrontare velocemente due sottostringhe qualunque di s.

2.1 Struttura

La struttura interna è molto simile a quella delle *sparse tables*. Infatti, un DBF è di fatto una matrice, in cui alla riga t e colonna i ($1 \leq i \leq |s|$), è presente l'intero $\text{DBF}_t[i]$, corrispondente alla sottostringa (fattore o *factor* in inglese) $s[i..i + 2^t - 1]$, dove l'indicizzazione è intesa ciclica[3].

In altre parole, alla riga t della matrice si trovano informazioni circa tutte le sottostringhe di lunghezza 2^t presenti nella stringa, detti *basic factors*. I valori sulle righe sono tutti numeri interi compresi tra 1 e $|s|$, e preservano l'ordine tra i fattori, a patto che questi siano della medesima lunghezza.

La costruzione del DBF viene interrotta quando i fattori superano lunghezza $|s|$, perciò il numero di righe è dell'ordine di $\log |s|$.

☞ **Nota sulla ciclicità della stringa**
Considerare la stringa come ciclica è un trucco molto comodo. Infatti, da una parte questo ci permette di progettare più facilmente certi algoritmi (per esempio, la ricerca della minima rotazione di una stringa), dall'altra questo non ci impedisce di fare *fallback* sulla versione non-ciclica, ad esempio considerando solo le posizioni "significative" di ogni riga t, ovvero $1 \leq i \leq |s| - 2^t + 1$.

[2] È comunque molto simile a quella presentata nel capitolo sui DBF.

[3] Quindi, per esempio, la sottostringa di lunghezza 4 che inizia alla posizione 3 della stringa `abacb` è `acba`.

▷ *Esempio:*

s	a	b	b	a	a	b	b	a	b	b	a
DBF_0	1	2	2	1	1	2	2	1	2	2	1
DBF_1	2	4	3	1	2	4	3	2	4	3	1
DBF_2	2	5	3	1	2	6	4	2	5	3	1
DBF_3	3	9	6	2	4	10	7	3	8	5	1

2.2 Costruzione

La costruzione di un DBF a partire da una stringa s è piuttosto facile. Infatti, il basic factor di lunghezza 2^t ($t > 0$) che inizia alla posizione i è ottenuto concatenando i due factor di lunghezza 2^{t-1} che iniziano alla posizione i e $i+2^{t-1}$ (gli indici sono da intendersi ciclici). Dal momento che possiamo supporre induttivamente che i fattori di lunghezza 2^{t-1} siano già stati confrontati, possiamo applicare radix sort sui fattori di lunghezza 2^t, visti come coppie sull'alfabeto dei factor di lunghezza 2^{t-1}. Questo algoritmo ha chiaramente complessità $|s| \log |s|$.

2.3 Confronto di sottostringhe

Per confrontare due sottostringhe **di pari lunghezza** s_1 e s_2, appartenenti alla stessa stringa s, ovvero stabilire se vale $s_1 < s_2$, $s_1 = s_2$ oppure $s_1 > s_2$, possiamo studiare i basic factors che le compongono.

☞ Stiamo tacitamente assumendo che le sottostringhe s_1 e s_2 di s ci vengano indicate implicitamente, ad esempio come coppie (posizione iniziale in s, lunghezza della sottostringa). Infatti, se ci fossero date esplicitamente, non avrebbe senso porsi la domanda di confrontarle velocemente, dal momento che per il solo fatto di leggerle o memorizzarle, staremmo già pagando un costo pari alla somma della loro lunghezza, equivalente al confronto "ingenuo".

In particolare, sia 2^t la più grande potenza di 2 minore o uguale a $n = |s_1| = |s_2|$, e definiamo:

$$s_{1L} = s_1[1..2^t], \quad s_{1R} = s_1[n - 2^t + 1..n]$$
$$s_{2L} = s_2[1..2^t], \quad s_{2R} = s_2[n - 2^t + 1..n]$$

Per confrontare s_1 con s_2 possiamo confrontare (lessicograficamente) le coppie (s_{1L}, s_{1R}) e (s_{2L}, s_{2R}): la relazione tra le coppie è identica alla relazione tra s_1 e s_2.

Pertanto, quando s_1 e s_2 hanno pari lunghezza, è possibile confrontare le sottostringhe in tempo costante. Rimane da capire come operare il confronto nel caso in cui s_1 e s_2 sono di diversa lunghezza. Senza perdita di generalità, possiamo assumere che sia $|s_1| < |s_2|$. Sfruttando nuovamente le proprietà dell'ordinamente lessicografico, possiamo restringere s_2 al suo prefisso di lunghezza pari ad $|s_1|$, e confrontare s_1 con la nuova sottostringa ottenuta s_2', con l'unica accortezza di gestire il caso $s_1 = s_2'$. Questo mostra che è possibile confrontare due sottostringhe qualsiasi di s in tempo costante.

☞ **Confronto con l'hashing**
Il metodo del DBF fornisce un criterio **esatto** per confrontare, in tempo costante, due sottostringhe arbitrarie s_1 e s_2 della stessa stringa. L'hashing fornisce un criterio **probabilistico** per stabilire l'uguaglianza tra due stringhe (ma non determina quale delle due è la minore), sempre in tempo costante.

Il metodo del DBF necessita di un preprocessing (offline) di costo $\mathcal{O}(|s| \log |s|)$ per una stringa di lunghezza $|s|$.

Vale la pena notare che è possibile risolvere i problemi presentati al capitolo sull'hashing usando il metodo del DBF:

- **Uniq**: concateniamo tutte le parole, separandole con un carattere speciale #. Calcoliamo il DBF della stringa ottenuta, e poi consideriamo solo le posizioni "interessanti", cioè quelle corrispondenti all'inizio di una parola dell'input.
- **Matching**: concateniamo il pattern e il testo, e usiamo l'operatore di confronto appena definito, per ogni sottostringa del testo di lunghezza pari a quella del pattern.

2.4 ♣ Ordinare tutte le sottostringhe di lunghezza fissata

Data una stringa (ciclica) s e un intero k, vogliamo ordinare tutte le sottostringhe di s di lunghezza k.

Come prima, dobbiamo lavorare implicitamente, sfruttando il fatto che sappiamo che le stringhe da trattare sono tutte sottostringhe di s. In particolare, sfruttiamo la relazione biunivoca tra la posizione $1 \leq i \leq |s|$ e la sottostringa $s[i..i+k-1]$. Introdotta la mappa $r : i \mapsto s[i..i+k-1]$, il problema si riduce a riordinare gli indici secondo r, cioè in modo tale che ogni volta che $r(i) \leq r(j)$, i compaia prima di j nella permutazione.

Usando l'operatore di confronto in tempo costante introdotto nella sezione precendente, possiamo applicare un generico algoritmo di ordinamento per confronti, ottenendo un algoritmo di complessità $\mathcal{O}(|s| \log |s|)$. Tuttavia possiamo fare di meglio, utilizzando un algoritmo di ordinamento non per confronti.

Ripercorrendo le tappe della sezione precedente, possiamo associare ad ogni $r(i)$ la mappa dei suoi due basic factors ricoprenti. Ci riduciamo ad avere $|s|$ coppie da dover ordinare, in cui ogni entry delle copie è un intero compreso tra 1 e $|s|$. Applicando radix sort, è possibile ordinare le tuple in tempo $\mathcal{O}(|s|)$. Infine, notiamo che è sufficiente interrompere la costruzione del DBF di s non appena i factors considerati eccedono lunghezza k. Pertanto, l'algoritmo considerato ha complessità $\mathcal{O}(|s| \log k)$.

2.5 ♣ Costruzione dei suffix array

Data una stringa (non ciclica) s, ordinare tutti i suoi suffissi.

Con il risultato della sezione precedente, è immediato risolvere il problema in questione. Infatti, concettualmente possiamo estendere s a destra con un carattere speciale $\#$, tale che sia valutato come minore di tutti i caratteri dell'alfabeto di s. Se ora ordiniamo tutte le sottostringhe di lunghezza $|s| + 1$ della stringa $(s + \#)$, otteniamo l'ordinamento che desideriamo. Questo algoritmo ha complessità $\mathcal{O}(|s| \log |s|)$.

☞ **Nota sull'uso di memoria**
Per costruire il suffix array di s è sufficiente considerare una sola riga del DBF, perciò non ha senso usare $\mathcal{O}(|s| \log |s|)$ memoria per memorizzare l'intera tabella.

2.6 ♣ Numero di sottostringhe distinte

Data una stringa s, contare quante sono le sottostringhe distinte.

La soluzione ingenua di questo problema ha complessità cubica in $|s|$. Possiamo tuttavia portare a quadratica la complessità sfruttando il risultato della sezione 2.4.

Infatti, per ogni $k = 1, \ldots, |s|$, possiamo riordinare tutte le sottostringhe di s di lunghezza k, filtrando poi quelle uguali (il controllo di uguaglianza tra sottostringhe richiede lavoro costante utilizzando il contronto tramite DBF). Sommando il numero di sottostringhe distinte di lunghezza k, su tutti i k possibili, otteniamo la risposta.

2.7 ♣ Massima sottostringa ripetuta

Data una stringa s, determinare una sottostringa di lunghezza massima che compare almeno k volte in s (le occorrenze possono sovrapporsi).

Sappiamo già come risolvere il problema decisionale "esiste una sottostringa di lunghezza n che compare almeno k volte in s?" in tempo $\mathcal{O}(|s|)$, a valle della creazione del DBF di s in tempo $\mathcal{O}(|s| \log |s|)$, utilizzando i risultati della sezione 2.4.

Notiamo inoltre che se s ammette una sottostringa di lunghezza $\tilde{n}$ ripetuta almeno k volte, allora la stessa cosa vale per ogni $n < \tilde{n}$. Questo ci suggerisce di usare ricerca binaria sul massimo valore di n. Notando che questo valore è sicuramente minore di $|s|$, abbiamo ottenuto un algoritmo di complessità $\mathcal{O}(|s| \log |s|)$.

2.8 ♣ Ricerca di palindromi

Data una stringa s, determinare il numero di sottostringhe palindrome di s.

Data una stringa s, determinare una sottostringa palindroma di lunghezza massima.

Le stringhe non possono essere cicliche in questo caso, perché in certi casi (*quali?*) le risposte potrebbero non essere ben definite.

Mostreremo come risolvere il secondo problema, cioè la determinazione della massima sottostringa palindroma. Prima di tutto conviene applicare una trasformazione della stringa $s = s_1\ s_2\ \cdots\ s_n$, portandola nella stringa

$$s' = s_1\ \#\ s_2\ \#\ \cdots\ \#\ s_n$$

Il vantaggio di questa trasformazione è che tutti i palindromi di s', in corrispondenza diretta con quelli di s, risultano di lunghezza dispari. Potremo allora soffermarci solo su questa classe di sottostringhe palindrome, che godono della (piacevole) proprietà di possedere un *centro* ben definito.

Implementeremo una funzione LPS(i) che, ricevuta una posizione i di s', ritorna la lunghezza della più lunga sottostringa palindroma *centrata in* i.

L'implementazione ingenua di LPS prova, per ogni possibile "raggio del palindromo" n, se $s'[i-n..i+n]$ è un palindromo. In tal caso diremo che n è un *raggio valido*. Il massimo raggio valido costituisce la risposta dell'istanza di LPS. Dal momento che n cresce di 1 ogni volta, la determinazione della "palindromicità" delle varie sottostringhe può essere operata in tempo costante (*come*?). L'algoritmo ingenuo ha una complessità $\mathcal{O}(|s|^2)$, essendo $|s'| \in \Theta(|s|)$.

Un'implementazione più furba sfrutta il fatto che se n **non** è un raggio valido, allora neanche $m > n$ lo è. Possiamo quindi pensare di fare ricerca binaria sul valore del massimo raggio valido. Questa volta, per controllare se una sottostringa è un palindromo dobbiamo cambiare strategia. A tal scopo, possiamo prima calcolare i DBF di s' e del rovescio di s', e operare un confronto delle due metà del potenziale palindromo[4]. Questo porta il controllo di palindromicità un'operazione logaritmica in $|s'|$, quindi anche in $|s|$. Questo algoritmo ha una complessità pari a $\mathcal{O}(|s| \log |s|)$.

3 Funzioni standard

Introduciamo ora due funzioni molto comuni, definite per ogni posizione di una stringa s. Gli algoritmi per calcolare questi valori saranno presentati più avanti, in un capitolo dedicato.

3.1 Funzione prefisso

Data una stringa $s = s_1\ s_2\ \cdots\ s_n$ e una posizione $1 < i \leq n$, definiamo $\textsc{pref}(i)$ come il massimo δ tale che $s[i..i+\delta-1] = s[1..\delta]$, con la ragionevole convenzione che $\textsc{pref}(i) = 0$ se $s[i] \neq s[1]$. Per la prima posizione della stringa, si preferisce definire $\textsc{pref}(1) = 0$, in quanto risulta più utile per molte applicazioni.

In altre parole, $\textsc{pref}(i)$ misura quanto è lungo il più lungo prefisso di $s[i..n]$ che è anche prefisso di s.

▷ *Esempio:*

s	a	b	b	a	a	b	b	a	b	b	a
PREF	-	0	0	1	4	0	0	4	0	0	1

È possibile costruire la tabella dei valori di $\textsc{pref}(i)$, $i = 1, \ldots, n$ in tempo lineare in $n = |s|$, come vedremo alla sezione 4.1.

3.2 Funzione bordo

Data una stringa $s = s_1\ s_2\ \cdots\ s_n$ e una posizione $1 \leq i \leq n$, definiamo $\textsc{bord}(i)$ come la dimensione del massimo bordo di $s[1..i]$, ovvero la dimensione del più grande suffisso proprio di $s[1..i]$ che sia anche prefisso di s.

[4] Questo passaggio sembra innocuo, ma in realtà è delicato. Ad esempio, come si può fare a garantire che i valori nei due DBF siano confrontabili? Spiegare.

▷ *Esempio:*

s	a	b	b	a	a	b	b	a	b	b	a
BORD	0	0	0	1	1	2	3	4	2	3	4

È possibile costruire la tabella dei valori di $\textsc{bord}(i)$, $i = 1, \ldots, n$ in tempo lineare in $n = |s|$, come vedremo alla sezione 4.2.

3.3 Relazione tra le due funzioni

È possibile ricostruire il vettore BORD a partire dal vettore PREF, in tempo lineare in $|s|$.

L'osservazione base su cui è costruita la conversione è la seguente:

- se $\textsc{pref}(i) = k$, allora $\textsc{bord}(j) \geq j - i + 1$ per $j = i, \ldots, i + k - 1$.
- se $\textsc{bord}(i) = k$, allora $\textsc{pref}(i - k + 1) \geq k$.

Mettendo assieme le due osservazioni, concludiamo che noto PREF, per ogni i il valore di $\textsc{bord}(i)$ è pari a $i - j^* + 1$, dove j^* è il minimo indice per cui valga $j^* + \textsc{pref}(j^*) - 1 \geq i$. Se tale indice non esiste, vale $\textsc{bord}(i) = 0$.

▷ *Dimostrazione:* per la prima osservazione, sicuramente deve essere $\textsc{bord}(i) \geq i - j^* + 1$. Supponiamo per assurdo che il valore corretto di $\textsc{bord}(i)$ sia pari a $k > i - j^* + 1$. Per la seconda osservazione deve essere allora $\textsc{pref}(i-k+1) \geq k$. D'altra parte $(i-k+1) < j^*$, e $(i-k+1)+\textsc{pref}(i-k+1)-1 \geq i-k+1+k-1 = i$. Questo contraddice la condizione su j^*, perciò si deduce che necessariamente $\textsc{bord}(i) = i - j^* + 1$.

Utilizzando un approccio "double pointer", è facile implementare l'algoritmo descritto in modo efficiente, ovvero con complessità lineare nella lunghezza dei vettori.

3.4 ♣ Stringhe periodiche

Data una stringa s, determinare la più corta stringa t per cui s si possa esprimere come concatenazione di una o più copie di t.

Chiamiamo "radice" di s una stringa t tale per cui s si può scrivere come concatenazione di una o più copie di t. Il problema chiede di determinare la più corta radice di s.

La stringa t deve necessariamente essere un prefisso di s, perciò possiamo concentrarci sul determinare quale sia la minima lunghezza di prefisso π tale per cui $s[\pi+1..|s|]$ ammetta $s[1..\pi]$ come sua radice.

Questa posizione corrisponde al minimo π per cui si ha

$$\textsc{pref}(\pi+1) = |s| - \pi.$$

Infatti, è immediato vedere che se la condizione è verificata allora il prefisso $s[1..\pi]$ è radice di $s[\pi+1..|s|]$. D'altra parte, se $s[1..\pi]$ è radice di $s[\pi+1..|s|]$ allora la condizione deve valere.

Possiamo quindi calcolare il vettore PREF e poi scandirlo alla ricerca della prima posizione π che rispetta la condizione. Questo algoritmo è lineare nella dimensione di s.

3.5 ♣ Matching

> Dato un pattern s e un testo t, listare tutte le occorrenze di s in t.

Come già accennato nell'ambito dei DBF, possiamo concatenare le stringhe s e t, ottenendo la stringa

$$S = s + \# + t.$$

Possiamo ora pensare di calcolare la funzione PREF di S. Le occorrenze di s in t corrispondono a tutte e sole le posizioni i di S per cui vale $\textsc{pref}(i) = |s|$.

3.6 ♣ Numero di sottostringhe distinte

> Data una stringa s, contare quante sono le sottostringhe distinte.

Possiamo sfruttare la funzione PREF per costruire induttivamente la risposta. Supponiamo infatti di aver determinato il numero di sottostringhe distinte di una generica stringa $\tilde{s}$, e di voler ora calcolare il numero di sottostringhe distinte della stringa $c + \tilde{s}$, dove c è un singolo carattere.

Aggiungere il carattere c a sinistra di $\tilde{s}$ crea $|\tilde{s}|+1$ sottostringhe potenzialmente nuove. Per determinare esattamente quante di queste non sono già presenti in $\tilde{s}$, possiamo calcolare il vettore $\textsc{pref}_{c+\tilde{s}}$ della stringa $c + \tilde{s}$. Il numero di sottostringhe nuove rispetto a $\tilde{s}$ è allora

$$|c + \tilde{s}| - \max_{i>1}\{\textsc{pref}_{c+\tilde{s}}(i)\}.$$

Questo algoritmo, come quello presentato alla sezione 2.6, è quadratico in $|s|$.

4 Calcolo di prefisso e bordo

4.1 Algoritmo Z

L'algoritmo Z si basa su un invariante: a mano a mano che iteriamo sulle posizioni della stringa, manteniamo un intervallo di posizioni $[L, R]$, detto *Z-box*, per cui vale che $s[L..R]$ è anche un prefisso di s, ovvero

$$s[L..R] = s[1..R - L + 1].$$

Inizialmente, porremo $L = R = 1$ per indicare che lo Z-box non è ancora stato calcolato. Supponiamo induttivamente di conoscere uno Z-box $[L, R]$ e di aver calcolato i valori di PREF per tutte le posizioni fino alla $i - 1$. Per determinare il valore $\text{PREF}(i)$ dobbiamo valutare diverse opzioni:

- Se $i > R$, lo Z-box non ci fornisce informazioni utili. Per questo ricalcoliamo uno Z-box, con estremo sinistro $L = i$ ed estremo destro calcolato nel modo ingenuo, cioè provando ad estendere R di un carattere alla volta, finché $s[L..R] = s[1..R - L + 1]$.
- Se $i \leq R$, sappiamo che $s[i..R] = s[i - L + 1..R - L + 1]$. Si pongono ora due casi:
  - Se $\text{PREF}(i-L+1) < R-i+1$, sicuramente $\text{PREF}(i) = \text{PREF}(i-L+1)$, e possiamo passare a concentrarci sulla posizione $i + 1$.
  - Se $\text{PREF}(i - L + 1) \geq R - i + 1$, allora sicuramente $\text{PREF}(i) \geq R - i + 1$, ma non conosciamo il valore esatto. Spostiamo allora lo Z-box corrente, assegnando $L = i$ ed estendendo R nel modo ingenuo.

In una singola passata, abbiamo quindi calcolato il valore di PREF per tutte le posizioni $i = 2, \ldots, |s|$. La complessità dell'algoritmo è lineare in $|s|$.

> ☞ **Algoritmo di Manacher**
> L'idea usata nello Z-algorithm, ovvero mantenere uno Z-box grazie al quale possiamo evitare di fare molti confronti, è abbastanza generale. L'algoritmo di Manacher, che permettere di stabilire la lunghezza $\text{PAL}(i)$ della massima sottostringa palindroma di s centrata nella posizione i, sfrutta questa idea. L'implementazione è in grado di determinare il valore PAL di tutte le posizioni della stringa in tempo lineare in $|s|$.

4.2 Algoritmo KMP

L'osservazione su cui si basa l'algoritmo KMP è la seguente: se $s[1..\pi]$ è un bordo di s, allora $s[1..\pi - 1]$ è un bordo di $s[1..|s| - 1]$.

Come prima, supponiamo di aver determinato il valore di BORD fino alla posizione $i-1$. Per l'osservazione, possiamo iterare su tutti i bordi $s[1..\pi]$ di $s[1..i-1]$, dal più lungo al più corto, finchè non è verificato che $s[1..\pi] + s[i]$ è bordo di $s[1..i]$.

☞ **Caratterizzazione dei bordi di una stringa**
Sia $\text{Bord}(s)$ la funzione che, presa una stringa s, ne ritorna il più lungo bordo (eventualmente la stringa vuota). Tutti e soli i bordi di s sono

$$\text{Bord}(s),\ \text{Bord}^2(s),\ \text{Bord}^3(s)\ldots,$$

dove la successione continua fino alla stringa vuota.

La precedente osservazione ci conduce ad un algoritmo estremamente compatto di complessità lineare in $|s|$, come rivela un semplice argomento ammortizzato.

■ Codice algoritmo Z

```
std::vector<size_t> z_algorithm(const std::string& s) {
    size_t n = s.length();
    std::vector<size_t> z(n);

    size_t L = 0, R = 0;
    for (size_t i = 1; i < n; ++i) {
        if (i <= R)
            z[i] = std::min(R - i + 1, z[i - L]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > R)
            L = i, R = i + z[i] - 1;
    }

    return std::move(z);
}
```

■ **Codice algoritmo KMP**

```cpp
std::vector<size_t> kmp(const std::string& s) {
    size_t n = s.length();
    std::vector<size_t> bord(n);

    for (size_t i = 1; i < n; ++i) {
        bord[i] = bord[i - 1];
        while (bord[i] && s[bord[i]] != s[i])
            bord[i] = bord[bord[i] - 1];
        if (s[bord[i]] == s[i])
            ++bord[i];
    }

    return std::move(bord);
}
```

5 Problemi aggiuntivi

♣ Stringception (MINSEQ)

Sono date due stringhe A e B ($1 \leq |A|, |B| \leq 10^5$). Si vuole inserire la stringa A all'interno della stringa B, cioè fissare un indice $0 \leq i \leq |B|$ e costruire la stringa $C_i = B[1..i] + A + B[i+1..|B|]$. Determinare un possibile valore di i che rende C_i la minima possibile, lessicograficamente.

♣ Selling RNA (JOI Open 2016)

Sono date n ($n \leq 100\,000$) stringhe $s_1, \ldots, s_n$ ($|s_1| + \cdots + |s_n| \leq 2 \cdot 10^6$) e m ($m \leq 100\,000$) query, ognuna della forma a_j, b_j ($\sum |a_j|, \sum |b_j| \leq 2 \cdot 10^6$). Per ogni query j, ritornare quante sono le stringhe s_i che ammettono a_j come prefisso e b_j come suffisso.