# Algoritmi di pattern matching su stringhe.

Livio Colussi

14 aprile 2016

# Introduzione

## Il problema del matching esatto

> Data una stringa $P$ detta *pattern* ed una stringa più lunga $T$ detta *testo* il problema del **matching esatto** consiste nel trovare tutte le occorrenze del pattern $P$ nel testo $T$.

Ad esempio se $P =$ `aba` e $T =$ `bbabaxababay` il pattern $P$ occorre in $T$ a partire dalle posizioni 3, 7 e 9. Osserviamo che due occorrenze di $P$ in $T$ possono sovrapporsi, come succede con le occorrenze di $P$ nelle posizioni 7 e 9.

## Importanza del problema del matching esatto

L'importanza pratica del problema del matching esatto è ovvia per chiunque usi un computer. Il problema sorge in talmente tante applicazioni che non è possibile elencarle tutte.

Alcune delle più comuni applicazioni si trovano nell'elaborazione di testi; in strumenti quali $grep$ di UNIX; in programmi di recupero di informazioni testuali quali Medline, Lexis e Nexis; nei programmi di ricerca nei cataloghi elettronici che hanno sostituito i cataloghi cartacei in molte grandi e medie biblioteche; nei motori di ricerca Internet che si muovono tra un enorme numero di testi presenti in rete per cercare materiale contenente certe parole chiave; nelle enormi biblioteche digitali che saranno disponibili nel prossimo futuro; nei giornali elettronici che già ora sono pubblicati on-line; negli elenchi telefonici on-line; nelle enciclopedie on-line ed altre applicazioni didattiche su CD-ROM; in basi di dati testuali specializzate quali quelle delle sequenze biologiche (RNA e DNA).

Benché l'importanza *pratica* del problema del matching esatto sia indubbia, ci si può chiedere se il problema sia anche di qualche interesse per la *ricerca* o per la *didattica*. Il problema del pattern matching esatto non è forse stato risolto così bene da poter essere racchiuso in una scatola nera e dato per scontato?

La risposta è che, per quanto riguarda le applicazioni tipiche di elaborazione di testi, non c'è più molto da fare. Ma la storia cambia radicalmente quando consideriamo altre applicazioni. Gli utenti di Melvyl, il catalogo on-line del sistema bibliotecario dell'Università della California sperimentano spesso dei lunghi e frustanti tempi di attesa anche per richieste molto semplici.

Anche l'operazione di *grepping* in una grande directory può dimostrare che il matching esatto non è ancora una operazione banale.

Se si usa CGC (una interfaccia molto popolare per effettuare ricerche in banche dati di DNA e di proteine) per ricercare in Genbank (la maggiore base dati di DNA) una stringa molto corta rispetto alla lunghezza delle stringhe che vengono ricercate nell'uso tipico di Genbank, la ricerca può richiedere anche qualche ora (magari soltanto per rispondere che la stringa non c'è).

Usando l'algoritmo di Boyer e Moore appositamente programmato per lavorare sul CD-ROM di Genbank (pensato come un unico lungo file di testo) il tempo di risposta si riduce a circa dieci minuti, nove dei quali utilizzati per spostare il testo da CD-ROM in memoria RAM e meno di uno per effettuare la ricerca vera e propria.

Certamente oggi ci sono programmi più efficienti per la ricerca su basi dati testuali (ad esempio BLAST) e vi sono macchine più veloci (ad esempio vi è un server per posta elettronica che gira su di un computer MasPar con 4000 processori).

Ma il fatto è che il problema del matching esatto non è così efficientemente e universalmente risolto da non necessitare di altra attenzione. Esso continua a rimanere un problema interessante perché le dimensioni delle basi dati continuano a crescere e perché il matching esatto continua ad essere un sottoproblema fondamentale in molti altri tipi complessi di ricerca che la gente continua ad inventarsi.

Ma forse la ragione più importante per studiare approfonditamente il matching esatto è quella di apprendere le *idee* che sono state sviluppate per risolverlo. Anche assumendo che il matching esatto sia stato risolto a sufficienza, l'intero campo dello string matching rimane vitale e aperto e l'istruzione che si ottiene studiando il matching esatto può risultare fondamentale per risolvere nuovi problemi. Questa istruzione riguarda tre aspetti: algoritmi specifici, stile generale di sviluppo degli algoritmi e tecniche di analisi e di dimostrazione.

# 1 Definizioni fondamentali per le stringhe

Un *alfabeto* è un insieme $\Sigma$ di *simboli* distinguibili detti *caratteri dell'alfabeto*. Un alfabeto può essere *finito* o *infinito* e usualmente è dotato di un *ordine totale*.

Una *stringa* $X$ è una successione finita di caratteri dell'alfabeto. Il numero di caratteri che costituiscono la successione viene detto *lunghezza* della stringa e viene indicato con $|X|$. La stringa di lunghezza zero viene detta *stringa nulla* ed è usualmente indicata con $\epsilon$.

I caratteri di una stringa $X$ di lunghezza $n$ sono indiciati con gli interi $1, \ldots, n$. Scriveremo pertanto

$$X = \mathtt{x_1} \ldots \mathtt{x_n}$$

Talvolta useremo anche la notazione vettoriale per le stringhe e scriveremo $X[i]$ invece di $\mathtt{x_i}$ e quindi

$$X = X[1] \ldots X[n]$$

quest'ultima notazione si può abbreviare in

$$X = X[1, n]$$

La *concatenazione* di due stringhe $X$ ed $Y$, che viene indicata con

$$X \cdot Y$$

è la successione di caratteri che si ottiene giustapponendo le due successioni di caratteri $X$ ed $Y$. Formalmente, se $X = \mathtt{x_1 x_2 \ldots x_n}$ ed $Y = \mathtt{y_1 y_2 \ldots y_m}$ allora

$$X \cdot Y = \mathtt{x_1 x_2 \ldots x_n y_1 y_2 \ldots y_m}$$

Naturalmente

$$X \cdot \epsilon = \epsilon \cdot X = X$$

e quindi $\epsilon$ è *l'elemento neutro* per l'operazione di concatenazione.

La potenza $k$-esima $X^k$ di una stringa $X$ è definita come la concatenazione di $k$ copie della stringa $X$. Naturalmente $X^0 = \epsilon$ e $X^1 = X$ per ogni stringa $X$.

Una *sottostringa* di una stringa $X$ è una qualsiasi stringa $Y$ tale che

$$X = Z \cdot Y \cdot W$$

per due opportune stringhe $Z$ e $W$.

Ogni terna

$$(Z, Y, W)$$

tale che $X = Z \cdot Y \cdot W$ si dice una *occorrenza* di $Y$ in $X$. Quando $X = Z \cdot Y \cdot W$ si dice anche che la stringa $Y$ *occorre* in $X$ in posizione $i = |Z| + 1$.

Dunque la stringa $Y$ è sottostringa della stringa $X$ se vi è almeno una occorrenza di $Y$ in $X$, ma naturalmente vi possono essere più occorrenze di $Y$ in $X$.

Spesso avremo bisogno di considerare distinte occorrenze diverse di una stessa sottostringa; in questo caso useremo la notazione vettoriale scrivendo $Y = X[i, j]$ per indicare la sottostringa $Y$ di lunghezza $j - i + 1$ che occorre in posizione $i$ nella stringa $X$; infatti $X = Z \cdot Y \cdot W = X[1, i-1]X[i, j]X[j+1, n]$.

In questo caso scriveremo inoltre

$$X[i, j] = X[k, l]$$

per indicare che $X[i, j]$ ed $X[k, l]$ sono occorrenze in $X$ di una stessa sottostringa

$$Y = X[i, j] = X[k, l]$$

Sia $X = Z \cdot Y \cdot W$ una stringa di lunghezza $n = |X|$ contenente una sottostringa $Y$ di lunghezza $m = |Y|$.

Se $Z = \epsilon$ si dice che $Y$ è un *prefisso* di $X$ ed in questo caso

$$Y = X[1, m]$$

Se $W = \epsilon$ si dice che $Y$ è un *suffisso* di $X$ ed in questo caso

$$Y = X[n - m + 1, n]$$

Se una stessa stringa $Y$ occorre nella stringa $X$ sia come prefisso che come suffisso si dice che $Y$ è un *bordo* di $X$ ed in questo caso

$$Y = X[1, m] = X[n - m + 1, n]$$

Se $Y$ è un bordo di $X$ allora esistono $Z$ e $W$ tali che

$$X = Z \cdot Y = Y \cdot W$$

In questo caso le due stringhe $Z$ e $W$ hanno la stessa lunghezza

$$p = |Z| = |W| = n - m$$

Alla lunghezza $p$ si dà il nome di *periodo* della stringa $X$. Questo nome è giustificato dal seguente Lemma.

**Lemma 1.1** *Sia $X = \mathtt{x_1 x_2 \ldots x_n}$ una stringa di lunghezza $n$ con un bordo $Y$ di lunghezza $m$ e sia $p = n - m$. Allora $\mathtt{x_i} = \mathtt{x_{i+p}}$ per ogni $i$ tale che $1 \leq i \leq n - p$. Viceversa, se $\mathtt{x_i} = \mathtt{x_{i+p}}$ per ogni $i$ tale che $1 \leq i \leq n - p$ allora $Y = X[1, n - p]$ è un bordo di $X$.*

**Dimostrazione.** Per definizione $Y$ è un bordo della stringa $X$ se e solo se

$$Y = X[1, m] = X[n - m + 1, n]$$

Ma $X[1, m] = X[n - m + 1, n]$ se e solo se sono uguali i corrispondenti caratteri $\mathtt{x_i}$ e $\mathtt{x_{i+n-m}}$ per $i = 1, \ldots, m$ e questo è come dire $\mathtt{x_i} = \mathtt{x_{i+p}}$ per ogni $i$ tale che $1 \leq i \leq n - p$ con $p = n - m$. □

Per il Lemma 1.1 bordi e periodi sono legati dalla relazione:

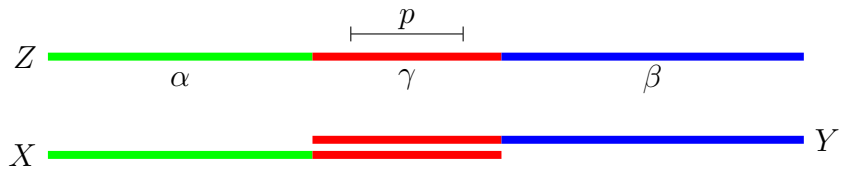
> La stringa $X$ di lunghezza $n$ ha un bordo $Y$ di lunghezza $m$ se e solo se $p = n - m$ è un periodo della stringa $X$.

Un prefisso, suffisso, bordo o sottostringa di una stringa $X$ si dice *proprio* se esso non è ne la stringa nulla ne l'intera stringa $X$. Un periodo $p$ di una stringa $X$ di lunghezza $n$ si dice *proprio* se $0 < p < n$. Notiamo che $p = 0$ e $p \geq n$ sono periodi (degeneri) di ogni stringa $X$ di lunghezza $n$.

Una stringa $X$ di lunghezza $n$ si dice *periodica* se ha un periodo $p$ tale che $0 < 2p \leq n$ (equivalentemente: un bordo di lunghezza $m$ tale che $m < n \leq 2m$).

Ci sarà anche utile il seguente lemma.

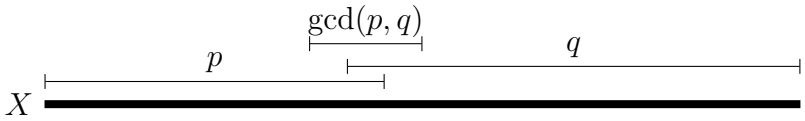
**Lemma 1.2** *Siano $X$ ed $Y$ due stringhe aventi lo stesso periodo $p$ e tali che un suffisso $\gamma$ di $X$ di lunghezza $|\gamma| \geq p$ sia anche prefisso di $Y$, ossia $X = \alpha\gamma$ ed $Y = \gamma\beta$. La stringa $Z = \alpha\gamma\beta$ ha anch'essa periodo $p$.*



**Dimostrazione.** Siano $\mathtt{z_i}$ e $\mathtt{z_{i+p}}$ due caratteri di $Z$ a distanza $p$. Siccome $|\gamma| \geq p$ tali due caratteri o stanno entrambi in $X = \alpha\gamma$ oppure entrambi in $Y = \gamma\beta$ e quindi sono uguali. □

Un risultato classico più sorprendente è il seguente Lemma di Periodicità che afferma che due periodi distinti $p$ e $q$ non possono coesistere troppo a lungo in una stessa stringa senza che la stringa abbia anche periodo $\gcd(p, q)$.

**Lemma 1.3 (Lemma di periodicità)** *Sia $X$ una stringa di lunghezza $n$ avente due periodi $p$ e $q$ non entrambi nulli. Se $n \geq p + q - \gcd(p, q)$ allora la stringa $X$ ha anche periodo $\gcd(p, q)$.*

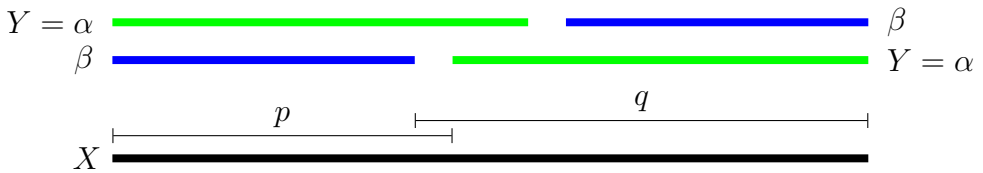



**Dimostrazione.** Assumiamo $p \leq q$. La dimostrazione è per induzione su $p + q$. Per la base dell'induzione: se $p + q = 1$ allora $p = 0$ e $q = \gcd(p, q) = 1$ e quindi $X$ ha periodo $\gcd(p, q)$.

Consideriamo quindi il caso $p + q > 1$. Se $p = 0$ oppure $p = q$ allora $\gcd(p, q) = q$ e dunque $X$ ha periodo $\gcd(p, q)$.

Supponiamo quindi $1 \leq p < q$.

La stringa $X$ ha bordi $\alpha$ e $\beta$ di lunghezze rispettive $n - p$ ed $n - q$. La stringa $\beta$ è anche bordo della stringa $Y = \alpha$ e di conseguenza $Y$ ha periodo $r = |\alpha| - |\beta| = q - p$.




Siccome $p + r < p + q$ e $\gcd(p, r) = \gcd(p, q)$ possiamo applicare l'ipotesi induttiva ad $Y$ la cui lunghezza è

$$|\alpha| = n - p \geq q - \gcd(p, q) = p + r - \gcd(p, r)$$

e che sappiamo avere periodi $p$ ed $r$. Dunque $Y$ ha periodo $\gcd(p, r) = \gcd(p, q)$.

Siccome inoltre

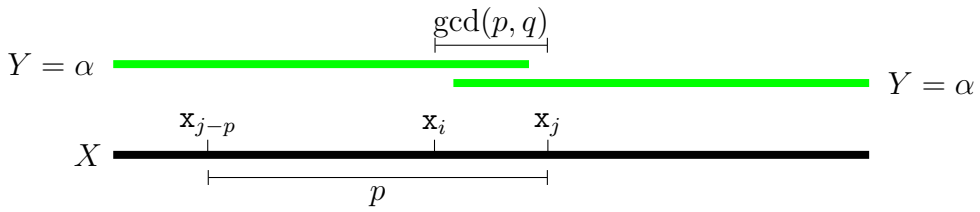
$$2|\alpha| = n - p + n - p \geq n - p + q - \gcd(p, q) \geq n$$

il prefisso $\alpha$ di $X$ ed il suffisso $\alpha$ di $X$ coprono tutto $X$.[1]

Consideriamo due caratteri $\mathtt{x}_i$ ed $\mathtt{x}_j$ della stringa $X$ a distanza $j - i = \gcd(p, q)$. Se entrambi i caratteri appartengono al prefisso $\alpha$ di $X$ oppure appartengono entrambi al suffisso $\alpha$ di $X$ allora $\mathtt{x}_i = \mathtt{x}_j$ poiché la stringa $Y = \alpha$ ha periodo $\gcd(p, q)$.

Altrimenti $\mathtt{x}_i$ appartiene al prefisso $\alpha$ di $X$ e $\mathtt{x}_j$ appartiene al suffisso $\alpha$ di $X$.

[1] Osserviamo che non si puòapplicare il Lemma 1.2 in quanto non è detto che prefisso e suffisso si sovrappongano per almeno $\gcd(p, q)$ caratteri. Sappiamo soltanto che essi coprono tutto $X$.




Siccome $\mathtt{x}_j$ appartiene al suffisso $\alpha$ di $X$ deve essere $j > n - |\alpha| = p$ e quindi $j - p \geq 1$ e possiamo considerare il carattere $\mathtt{x}_{j-p}$ che, poiché $p \geq \gcd(p, q)$, precede il carattere $\mathtt{x}_i$ e quindi appartiene anch'esso al prefisso $\alpha$ di $X$.

La distanza tra $\mathtt{x}_{j-p}$ e $\mathtt{x}_i$ è $p - \gcd(p, q)$ che è un multiplo di $\gcd(p, q)$. Poiché $\mathtt{x}_{j-p}$ e $\mathtt{x}_i$ appartengono entrambi al prefisso $\alpha$ di $X$ che sappiamo avere periodo $\gcd(p, q)$ essi sono uguali e quindi, anche in questo caso, $\mathtt{x}_i = \mathtt{x}_{j-p} = \mathtt{x}_j$. Pertanto la stringa $X$ ha periodo $\gcd(p, q)$. □

### Confusione nella terminologia

Le parole "stringa" e "parola" sono spesso considerati sinonimi nella letteratura informatica. Noi useremo sempre "stringa" e lasceremo a "parola" il significato colloquiale di parola di senso compiuto di una lingua parlata. Peggio ancora "stringa" e "sequenza" sono spesso assunti come sinonimi. Questo è fonte di confusione in quanto "*sotto*stringa" e "*sotto*sequenza" sono concetti completamente diversi. I caratteri di una sottostringa di una stringa $X$ devono comparire *consecutivamente* in $X$ mentre i caratteri di una sottosequenza possono essere intervallati con caratteri che non stanno nella sottosequenza.

## 2 Primi algoritmi e preelaborazione del pattern

### L'algoritmo ingenuo

Tutte le trattazioni del matching esatto iniziano con il *metodo ingenuo*. Il metodo ingenuo allinea il primo carattere del pattern $P$ con il primo carattere del testo $T$, confronta quindi i caratteri di $P$ e di $T$ da sinistra a destra finché o si arriva alla fine di $P$, nel qual caso viene segnalata una occorrenza di $P$ in $T$, oppure vengono incontrati due caratteri diversi (un mismatch). In entrambi i casi $P$ viene quindi spostato avanti di un posto e si ricominciano i confronti a partire dal primo carattere di $P$ con il carattere del testo ad esso allineato. Il procedimento viene ripetuto finché l'ultimo carattere di $P$ non sorpassi l'ultimo carattere di $T$.

Ad esempio, l'algoritmo ingenuo applicato alle due stringhe $P =$ `abcdabce` e $T =$ `cabcdabcdabce` opera come in Figura 1, dove il colore verde di un carattere del pattern indica un confronto con un carattere uguale del testo (e in questo caso si dice che vi è un *match*) mentre il colore rosso indica un confronto con un carattere diverso del testo (e in tal caso si dice che vi è un *mismatch*). I caratteri del pattern di colore bianco non vengono confrontati.

Formalmente, l'algoritmo ingenuo è il seguente.

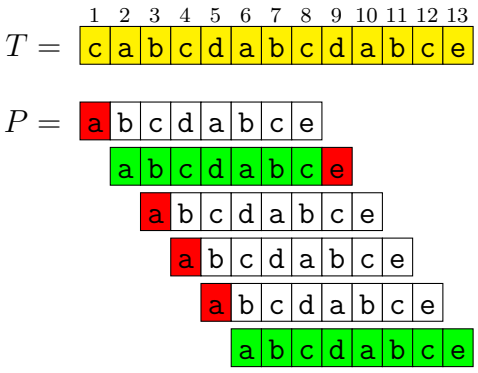



Figura 1: L'algoritmo ingenuo applicato alle due stringhe $P =$ `abcdabce` e $T =$ `cabcdabcdabce`. I match sono colorati in verde mentre i mismatch sono colorati in rosso. In totale vengono effettuati 20 confronti.

INGENUO$(P, T)$

```
1  // T testo di lunghezza n e P pattern di lunghezza m ≤ n.
2  for i = 1 to n − m + 1
3       j = 1
4       while j ≤ m and P[j] == T[i + j − 1]
5            j = j + 1
6       if j > m
7            "segnala l'occorrenza in posizione i"
```

Se $P = \mathtt{a}^m$ e $T = \mathtt{a}^n$ sono stringhe costituite dallo stesso carattere `a` ripetuto $m$ ed $n$ volte rispettivamente, vi è una occorrenza di $P$ in ognuna delle prime $n - m + 1$ posizioni di $T$, ed il metodo effettua esattamente $m(n - m + 1)$ confronti. Pertanto il numero di confronti operati da questo algoritmo nel caso pessimo è $\Theta(mn)$.

Il metodo ingenuo è certamente semplice da capire e da programmare ma il suo tempo calcolo $\Theta(mn)$ nel caso pessimo è alquanto insoddisfacente e deve essere migliorato.

Non solo nel caso pessimo ma anche nella pratica il metodo ingenuo risulta troppo inefficiente per pattern e testi lunghi.

Negli anni '70 sono state sviluppate diverse idee per migliorare l'efficienza dell'algoritmo ingenuo. Il risultato fu la riduzione del tempo calcolo nel caso pessimo da $\Theta(mn)$ a $\Theta(m + n)$.

## Come rendere più efficiente l'algoritmo ingenuo

Una prima semplificazione dell'algoritmo ingenuo si ottiene aggiungendo al pattern un carattere finale diverso da tutti i caratteri del pattern e del testo (la *sentinella*). Questo rende inutile il test $j \leq m$ nel ciclo $while$ e riduce il tempo calcolo richiesto per eseguire l'algoritmo (di circa $\frac{1}{3}$) ma non riduce la sua complessità asintotica.

INGENUO-CON-SENTINELLA$(P, T)$

```
1  // T testo di lunghezza n e P pattern di lunghezza m ≤ n.
2  P[m + 1] = $, T[n + 1] = x
3  for i = 1 to n − m + 1
4       j = 1
5       while P[j] == T[i + j − 1]
6            j = j + 1
7       if j > m
8            "segnala l'occorrenza in posizione i"
```

Le prime idee per rendere asintoticamente più efficiente l'algoritmo ingenuo consistono tutte nel cercare di spostare $P$ di più di una posizione quando si incontra un mismatch tra due caratteri corrispondenti, ma senza spostarlo così tanto da perdere qualche occorrenza di $P$ in $T$.

Spostando il pattern di più di una posizione si risparmiano confronti in quanto $P$ si muove più velocemente su $T$.

Oltre a spostare più velocemente il pattern $P$ alcuni metodi cercano di risparmiare confronti saltando, dopo aver spostato il pattern, i confronti dei caratteri di una parte iniziale del pattern.

Più avanti tutte queste idee verranno esaminate nel dettaglio. Per ora ci limiteremo soltanto a dare un'idea intuitiva di questi possibili miglioramenti riconsiderando l'esempio usato per illustrare l'algoritmo ingenuo in cui il pattern era $P =$ `abcdabce` ed il testo era $T =$ `cabcdabcdabce`.

L'algoritmo ingenuo richiede 20 confronti dei quali 5 sono mismatch e 15 sono match. Un algoritmo più furbo si sarebbe dovuto accorgere, dopo aver effettuato i primi 9 confronti, che le tre successive posizioni del pattern non possono essere occorrenze.

Infatti i match precedenti dicono che i caratteri del testo in tali posizioni sono uguali rispettivamente al secondo, terzo e quarto carattere del pattern e questi caratteri sono tutti e tre diversi dal primo carattere `a` del pattern.

Dopo i primi 9 confronti l'algoritmo furbo dovrebbe quindi spostare il pattern non di una sola posizione ma di quattro posizioni nel modo indicato in Figura 2. Questo riduce

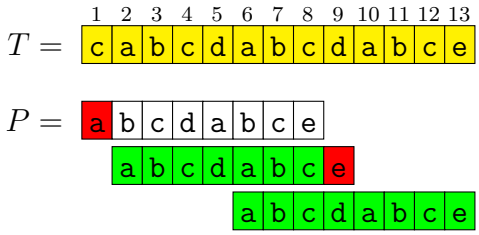



Figura 2: Un algoritmo un po' più furbo che effettua soltanto 17 confronti.

il numero di confronti a 17.

Un algoritmo ancora più furbo si sarebbe dovuto accorgere inoltre che dopo lo spostamento del pattern di quattro posizioni i primi tre caratteri del pattern `abc` sono uguali

ai corrispondenti tre caratteri del testo. Infatti i precedenti match assicurano che tali tre caratteri del testo sono uguali al quinto, sesto e settimo carattere del pattern che a loro volta sono uguali ai primi tre caratteri del pattern.

L'algoritmo ancora più furbo, dopo aver spostato il pattern di 4 posizioni, dovrebbe quindi cominciare a confrontare i caratteri del pattern con i corrispondenti caratteri del testo partendo non dal primo ma dal quarto (come indicato in Figura 3). Questo riduce il numero di confronti a 14.

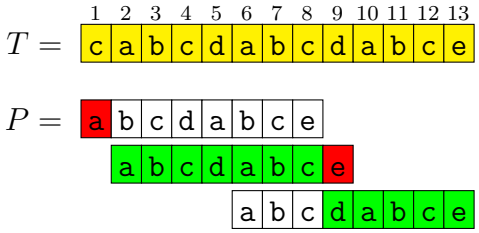



Figura 3: Un algoritmo ancora più furbo che effettua soltanto 14 confronti.

Non è però chiaro come si possano realizzare efficientemente queste idee. Implementazioni efficienti di queste idee sono state realizzate in un certo numero di algoritmi quali quello di Knuth, Morris e Pratt, una sua estensione che opera in tempo reale, l'algoritmo di Boyer e Moore ed altri successivi.

Tutti questi algoritmi si possono programmare in modo da richiedere tempo lineare $O(m+n)$.

## L'approccio della preelaborazione

Molti algoritmi di matching su stringhe riescono a risparmiare confronti nella fase di ricerca spendendo dapprima un tempo "modesto" per imparare qualcosa sulla struttura interna o del pattern o del testo.

Questa parte dell'algoritmo viene detta fase di *preelaborazione*. Durante la preelaborazione l'altra stringa può non essere ancora nota all'algoritmo.

La preelaborazione è seguita da una fase di *ricerca*, in cui le informazioni trovate durante la fase di preelaborazione vengono sfruttate per ridurre il lavoro da fare durante la ricerca delle occorrenze di $P$ in $T$.

Nell'esempio dell'algoritmo ingenuo si assume che la versione più furba sappia che il carattere `a` non ricompare prima della posizione 5 del pattern, e che la versione ancora più furba sappia che la sequenza `abc` si ripete a partire dalla posizione 5. Questa conoscenza si assume sia stata ottenuta durante la fase di preelaborazione.

Tutti gli algoritmi di matching esatto che vedremo preelaborano il pattern $P$. L'approccio opposto di preelaborare il testo viene usato per altri tipi di problemi.

Questi due tipi di preelaborazione sono "simili in spirito" ma spesso molto diversi nei dettagli e nella difficoltà concettuale.

Metteremo in evidenza la similarità discutendo dapprima una *preelaborazione fondamentale* che verrà poi usata per spiegare i dettagli specifici di tutti i metodi particolari.

## La preelaborazione fondamentale

Descriveremo la preelaborazione fondamentale per una generica stringa $S$ di lunghezza $n$. Nelle applicazioni specifiche la stringa $S$ può essere sia il pattern $P$ che il testo $T$.

**Definizione 2.1 (Funzione Prefisso)** *Data una stringa $S$ di lunghezza $n$ definiamo $\pi_i^S$ come la lunghezza del più lungo prefisso di $S$ che occorre in posizione $i$ in $S$.*

Quindi $\pi_i^S$ è il massimo $h$ tale che

$$S[1,h] = S[i, i+h-1]$$

ossia esso è la lunghezza del *più lungo prefisso comune* tra $S$ ed il suo suffisso $S[i,n]$.

Ad esempio se $S =$ `aabcaabdaae` allora

| $i$ | $\pi_i^S$ | più lungo prefisso comune |
|---|---|---|
| 1 | 11 | `aabcaabdaae` |
| 2 | 1 | `a` |
| 3 | 0 | $\epsilon$ |
| 4 | 0 | $\epsilon$ |
| 5 | 3 | `aab` |
| 6 | 1 | `a` |
| 7 | 0 | $\epsilon$ |
| 8 | 0 | $\epsilon$ |
| 9 | 2 | `aa` |
| 10 | 1 | `a` |
| 11 | 0 | $\epsilon$ |

Notiamo che $\pi_1^S = n$ per ogni stringa $S$ di lunghezza $n$ poiché il più lungo prefisso comune tra $S$ ed $S[1,n] = S$ è tutto $S$.

Quando la stringa $S$ è chiara dal contesto useremo $\pi_i$ invece di $\pi_i^S$.

Per definizione $S[i, i+\pi_i - 1]$ è una occorrenza in $S$ del prefisso $S[1,\pi_i]$. Questo significa che la stringa

$$Y = S[i, i+\pi_i - 1] = S[1,\pi_i]$$

è un bordo del prefisso $S[1, i+\pi_i - 1]$ di $S$ e, di conseguenza, $p = i-1$ è un periodo di tale prefisso.

Per $i = 1$ abbiamo $Y = S$ e $p = 0$ che sono bordo e periodo degeneri di $S$.

Se $i \geq 2$ allora o $i + \pi_i - 1 = n$, nel qual caso $Y = S[i,n]$, oppure

$$S[1+\pi_i] \neq S[i+\pi_i]$$

e, in entrambi i casi, la stringa $S[1, i+\pi_i - 1]$ è il più lungo prefisso di $S$ avente periodo $p = i-1$.

Le sottostringhe $S[i, i+\pi_i - 1]$ non sono necessariamente disgiunte ma possono sovrapporsi.

Dato un $i \geq 2$ consideriamo tutte le sottostringhe $S[j, j+\pi_j - 1]$ con $2 \leq j \leq i$ ed indichiamo con $r_i$ il valore massimo dell'estremo destro $j+\pi_j - 1$ di tali sottostringhe e

con $\ell_i = j$ l'estremo sinistro corrispondente. Se ci sono più sottostringhe $S[j, j + \pi_j - 1]$ con lo stesso estremo destro $r_i = j + \pi_j - 1$ massimo prendiamo l'estremo sinistro $\ell_i = j$ di una qualsiasi di esse.

Detto in altro modo: $S[1, r_i]$ è il più lungo prefisso di $S$ che ha un periodo $1 \leq p < i$ ed $\ell_i = p + 1$. Nel caso in cui $S[1, r_i]$ abbia più di un periodo proprio minore di $i$, scegliamo $\ell_i = p + 1$ dove $p$ è uno qualsiasi di tali periodi.

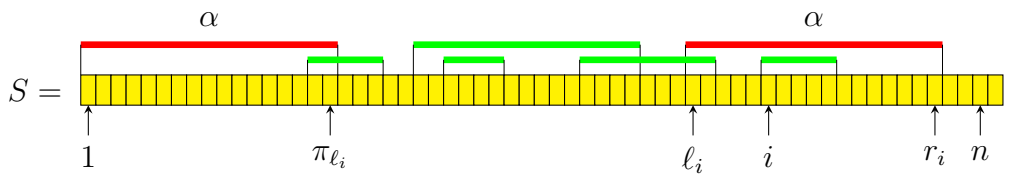



Figura 4: Alcune occorrenze di prefissi di $S$ che iniziano tra le posizioni 2 e $i$. Come si può osservare le occorrenze non sono disgiunte ma possono intersecarsi o essere contenute una nell'altra. L'occorrenza che termina più a destra è evidenziata in rosso; essa inizia nella posizione $\ell_i \leq i$, termina nella posizione $r_i$, ha lunghezza $\pi_{\ell_i}$ ed è uguale al prefisso $\alpha$ di $S$ di pari lunghezza.

Quindi $S[\ell_i, r_i]$ è la sottostringa di $S$ che termina più a destra tra tutte quelle che sono uguali ad un prefisso di $S$ e che iniziano in posizioni comprese tra 2 ed $i$. Una possibile situazione è illustrata in Figura 4 dove sono evidenziate in verde alcune occorrenze di prefissi di $S$ che iniziano tra le posizioni 2 e $i$. Tra esse l'occorrenza che termina più a destra è evidenziata in rosso; essa inizia nella posizione $\ell_i$, termina nella posizione $r_i$, ha lunghezza $\pi_{\ell_i}$ ed è uguale al prefisso $\alpha$ di $S$ di pari lunghezza.

Ad esempio, se

$$S = \texttt{aabaabcadaabaabce}$$

il massimo estremo destro di $S[j, j + \pi_j - 1]$ con $2 \leq j \leq 15$ si ha per $j = 10$ dove la funzione prefisso vale $\pi_{10} = 7$ per cui

$$r_{15} = 10 + \pi_{10} - 1 = 16$$

ed $\ell_{15} = 10$.

Invece, per $i = 3$, il massimo estremo destro di $S[j, j + \pi_j - 1]$ con $2 \leq j \leq 3$ si ha per $j = 3$ dove la funzione prefisso vale $\pi_3 = 0$ per cui

$$r_3 = 3 + \pi_3 - 1 = 2$$

ed $\ell_3 = 3$. Osserviamo che in quest'ultimo caso $\ell_i = r_i + 1 > r_i$.

## Preelaborazione fondamentale in tempo lineare

Mostreremo come calcolare tutti i valori $\pi_i$ per una stringa $S$ di lunghezza $n$ in tempo $O(n)$. Un approccio diretto basato sulla definizione richiederebbe tempo $O(n^2)$.

Per ragioni tecniche supponiamo che $S$ termini con un carattere diverso da tutti i precedenti (alla bisogna possiamo sempre aggiungere, alla fine di $S$, un carattere *sentinella* diverso da tutti i precedenti).

L'algoritmo di preelaborazione pone $\pi_1 = n$ e poi calcola successivamente i valori di $\pi_i$, $r_i$ ed $\ell_i$ per $i = 2, \ldots, n$.

Esso memorizza in un array $pref[1\,..\,n]$ i valori di $\pi_i$, ma per eseguire l'iterazione $i$-esima usa soltanto i valori di $r_{i-1}$ ed $\ell_{i-1}$; i valori precedenti non servono più.

L'algoritmo usa quindi una singola variabile $r$ per memorizzare i successivi valori di $r_i$ ed una singola variabile $\ell$ per memorizzare i successivi valori di $\ell_i$.

L'algoritmo calcola dapprima direttamente il valore di $\pi_2$ confrontando da sinistra a destra i caratteri di $S[2, n]$ con i caratteri di $S$ finché non trova un mismatch (deve trovarlo a causa della sentinella).

Il valore di $\pi_2$ è allora la lunghezza del massimo prefisso comune a tali due stringhe, ad $r = r_2$ viene assegnato il valore $\pi_2 + 1$ e ad $\ell = \ell_2$ viene assegnato il valore 2.

Assumiamo ora induttivamente di aver calcolato correttamente i valori di $\pi_j$ per ogni $j = 2, \ldots, i-1$ ed i valori di $r = r_{i-1}$ e di $\ell = \ell_{i-1}$.

Per calcolare $\pi_i$, $r = r_i$ ed $\ell = \ell_i$ consideriamo i seguenti casi e sottocasi.

**Caso 1:** Se $i > r$ si calcola direttamente $\pi_i$ confrontando da sinistra a destra i caratteri delle due stringhe $S[i, n]$ ed $S$ finché si trova un mismatch. Il valore di $\pi_i$ è allora uguale alla lunghezza $h$ del massimo prefisso comune. Si pone inoltre $r = i + \pi_i - 1$ ed $\ell = i$;

**Caso 2:** Se $i \leq r$ il carattere $S[i]$ ècontenuto nella sottostringa $\alpha = S[\ell, r]$ che per definizione di $\pi_\ell$ ed $r$, è il prefisso $\alpha = S[1, \pi_\ell]$ della stringa $S$.

Pertanto il carattere $S[i]$ compare anche nella posizione $i' = i - \ell + 1$ di $S$ e per la stessa ragione la sottostringa $\beta = S[i, r]$ deve comparire anche in posizione $i'$, ossia $\beta = S[i', \pi_\ell]$ come illustrato in Figura 5.

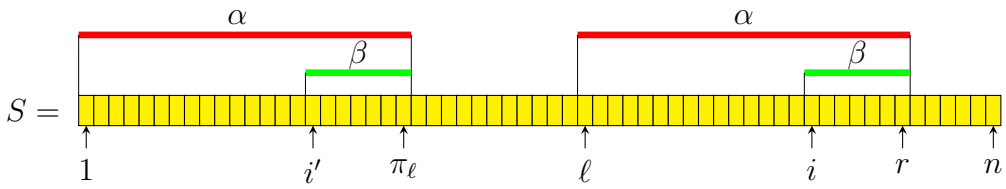



Figura 5: Se $i \leq r$ la stringa $\beta = S[i, r]$ compare anche alla fine del prefisso $\alpha$ e quindi $\beta = S[i', \pi_\ell]$.

In $i'$ occorrerà un prefisso $\gamma$ di $S$ di lunghezza $\pi_{i'}$: questo prefisso occorrerà a partire dalle posizioni 1 e $i'$ e, per la parte di lunghezza minore o uguale alla lunghezza di $\beta$, anche dalla posizione $i$ (dato che o $\beta$ contiene $\gamma$ o $\beta$ contenuto in $\gamma$).

Ne consegue che $S$ ed il suo suffisso $S[i, n]$ che inizia in posizione $i$ hanno un prefisso comune di lunghezza uguale al *minimo* tra $\pi_{i'}$ e $|\beta| = r - i + 1$.

Consideriamo i due sottocasi separatamente.

**Caso 2a:** Se $\pi_{i'} < |\beta|$ allora $\pi_i = \pi_{i'}$ ed $r$ ed $\ell$ rimangono invariate. La situazione è illustrata in Figura 6.

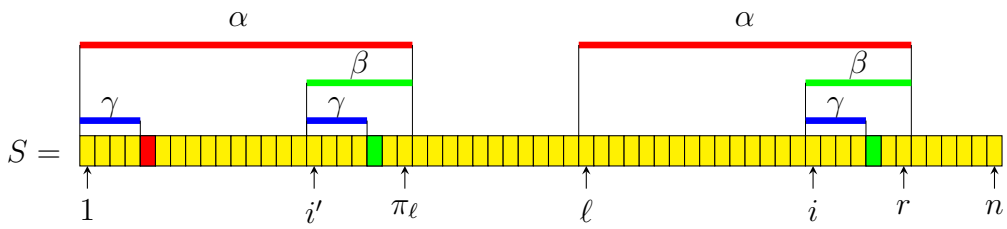



Figura 6: Se $\pi_{i'} < |\beta|$ il prefisso $\gamma$ di lunghezza $\pi_{i'}$ compare anche nelle posizioni $i'$ ed $i$ ed il carattere successivo (in rosso) è diverso dai due caratteri (in verde) successivi alle due occorrenze (che sono uguali tra loro). Quindi $\pi_i = \pi_{i'}$.

**Caso 2b:** Se $\pi_{i'} \geq |\beta|$ allora l'intera sottostringa $S[i, r]$ deve essere un prefisso di $S$ e quindi $\pi_i \geq |\beta| = r - i + 1$. L'algoritmo calcola quindi la lunghezza

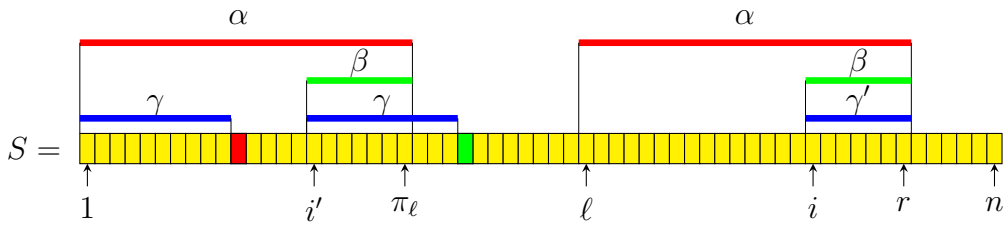



Figura 7: Se $\pi_{i'} \geq |\beta|$ il prefisso $\gamma$ di lunghezza $\pi_{i'}$ compare anche nella posizione $i'$ ma nella posizione $i$ compare soltanto la parte $\gamma' = \beta$ contenuta in $\alpha$. Per calcolare $\pi_i$ occorre quindi controllare direttamente i caratteri successivi.

$h$ del massimo prefisso comune tra $S$ ed $S[i, n]$ cominciando a confrontare i due caratteri in posizione $|\beta| + 1$ e $i + |\beta| = r + 1$ e prosegue finché non trova un mismatch. A questo punto $\pi_i$ viene posto uguale ad $h$ e si pone inoltre $r = i + \pi_i - 1$ ed $\ell = i$. La situazione è illustrata in Figura 7.

L'algoritmo di preelaborazione completo è il seguente.

FUNZIONE-PREFISSO($S$)

```
1  // S stringa di lunghezza n > 1 con sentinella alla fine.
2  pref[1] = n
3  h = 0
4  while S[1 + h] == S[2 + h]
5      h = h + 1
6  pref[2] = h, ℓ = 2, r = 2 + h − 1
7  for i = 3 to n
8      if r < i                                          // Caso 1
9          h = 0
10         while S[1 + h] == S[i + h]
11             h = h + 1
12         pref[i] = h, ℓ = i, r = i + h − 1
13     elseif pref[i − ℓ + 1] < r − i + 1                // Caso 2a
14         pref[i] = pref[i − ℓ + 1]
15     else                                              // Caso 2b
16         h = r − i + 1
17         while S[1 + h] == S[i + h]
18             h = h + 1
19         pref[i] = h, ℓ = i, r = i + h − 1
20 return pref
```

Nella Figura 8 sono riportati i confronti tra caratteri effettuati dall'algoritmo precedente quando esso viene eseguito con la stringa $S =$ `abaaabaabaaabc`.

**Teorema 2.2** *L'algorimo* FUNZIONE-PREFISSO *calcola correttamente i valori* $pref[i] = \pi_i$ *e aggiorna correttamente* $r$ *ed* $\ell$.

**Dimostrazione.** Per $i = 1$ l'algoritmo pone $pref[1] = \pi_1 = n$ e per $i = 2$ calcola $h = \pi_2$ direttamente e pone correttamente $pref[2] = h$, $r = r_2 = 2 + h - 1$ ed $\ell = \ell_2 = 2$.

Assumiamo quindi, quale invariante del ciclo **for**, che l'algoritmo abbia calcolato correttamente $pref[j] = \pi_j$ per ogni $j < i$, $r = r_{i-1}$ ed $\ell = \ell_{i-1}$ e dimostriamo che, dopo aver eseguita l'istruzione interna a tale ciclo, l'algoritmo ha calcolato correttamente anche $pref[i] = \pi_i$, $r = r_i$ ed $\ell = \ell_i$.

Nel Caso 1, $pref[i] = \pi_i$ viene calcolato direttamente ed è quindi corretto. Inoltre, siccome $i > r$ prima di calcolare $pref[i]$, non ci sono sottostringhe di $S$ uguali ad un suo prefisso che iniziano in posizioni comprese tra 2 e $i - 1$ e che finiscono in posizione $i$ o oltre.

La sottostringa $S[i, i+\pi_i-1]$ è prefisso di $S$, inizia in posizione $i$ e finisce in posizione $i + \pi_i - 1 \geq r$ ed è quindi corretto porre $r = r_i = i + \pi_i - 1$ ed $\ell = \ell_i = i$. L'algoritmo si comporta quindi correttamente nel Caso 1.

Nel Caso 2a, il massimo prefisso comune tra $S[i, n]$ ed $S$ non può essere più lungo di $\pi_{i'} < |\beta|$ altrimenti anche i due caratteri successivi $S[i + \pi_{i'}]$ ed $S[1 + \pi_{i'}]$ dovrebbero essere uguali. Ma il carattere $S[i + \pi_{i'}]$ è uguale al carattere $S[i' + \pi_{i'}]$ (perché $\pi_{i'} < |\beta|$) e quindi il carattere $S[i' + \pi_{i'}]$ sarebbe uguale al carattere $S[1 + \pi_{i'}]$ contraddicendo la definizione di $\pi_{i'}$.

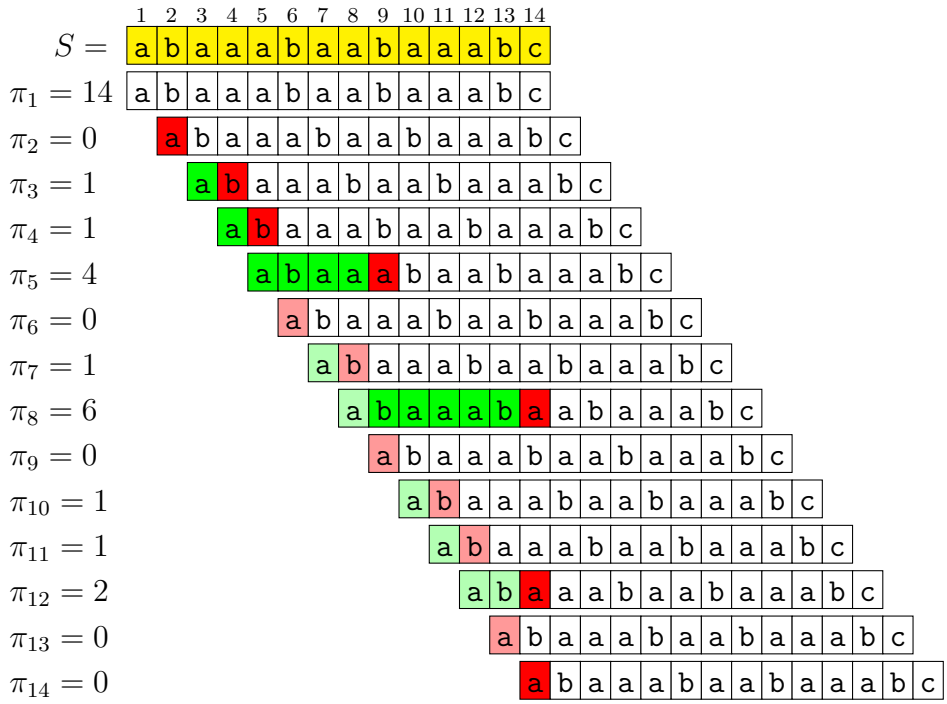



Figura 8: Confronti effettuati dall'algoritmo FUNZIONE-PREFISSO con input la stringa $S =$ `abaaabaabaaabc`. I match sono di colore verde e i mismatch sono di colore rosso mentre i colori verde chiaro e rosa indicano match e mismatch che l'algoritmo FUNZIONE-PREFISSO risparmia rispetto ad una versione ingenua che calcoli direttamente $\pi_i$ per ogni $i = 2, \ldots, n$. Si noti che FUNZIONE-PREFISSO ha al più un solo match per ogni carattere del testo ed ha al più un solo mismatch per ogni allineamento del pattern sul testo.

Pertanto in questo caso è giusto porre $pref[i] = pref[i']$. Infine $i + \pi_i - 1 < r$ e quindi $r$ ed $\ell$ devono rimanere invariati.

Nel Caso 2b, $\beta$ è prefisso comune di $S[i, n]$ ed $S$ e siccome l'estensione di tale prefisso comune viene verificata direttamente confrontando da sinistra a destra i caratteri successivi a partire dalle due posizioni $|\beta| + 1$ ed $r + 1 = i + |\beta|$, il valore di $pref[i] = \pi_i$ viene calcolato correttamente. Infine siccome $i + \pi_i - 1 \geq r$ l'algoritmo aggiorna correttamente i valori di $r$ e di $\ell$. □

**Teorema 2.3** *L'algoritmo* FUNZIONE-PREFISSO *calcola tutti i valori di* $\pi_i$ *in tempo* $O(n)$.

**Dimostrazione.** Il tempo è proporzionale al numero di iterazioni del ciclo **for**, che sono $n-2$, più il numero di confronti tra caratteri che vengono effettuati nei test dei cicli **while**.

Il primo ciclo **while**, quello esterno al ciclo **for**, calcola $\pi_2$ e per ciascuno dei valori successivi dell'indice $i = 3, \ldots, n$ il calcolo di $\pi_i$ richiede l'esecuzione di al più un solo ciclo **while** interno al ciclo **for**.

Ogni esecuzione di un ciclo **while** termina non appena trova un mismatch. Quindi vengono effettuati in totale al più $n - 1$ confronti con esito negativo.

Rimangono da considerare i confronti con esito positivo. Ogni confronto coinvolge due caratteri $S[1+h]$ ed $S[i+h]$ ($S[2+h]$ nel primo ciclo) che chiameremo rispettivamente carattere *sinistro* e carattere *destro*.

Dopo ogni confronto con esito positivo entrambi i caratteri vengono spostati a destra di una posizione.

Sia $S[i+h]$ il carattere destro quando termina uno qualsisi dei cicli **while**. In questo caso viene posto $r = i+h-1$ per cui la posizione di tale carattere è $S[r+1]$.

Il successivo ciclo **while** inizia con carattere destro $S[i]$ se $i > r$ (ossia $i \geq r+1$), altrimenti inizia con carattere destro $S[i+h]$ con $h = r-i+1$ (ossia con il carattere $S[r+1]$). In entrambi i casi il carattere destro non si sposta mai a sinistra durante tutta l'esecuzione dell'algoritmo.

Quindi vengono eseguiti al più $n-1$ confronti con esito positivo e pertanto il numero totale di confronti è minore o uguale di $2n-2$. □

## Il più semplice algoritmo di matching esatto in tempo lineare

Prima di usare la preelaborazione fondamentale per altre preelaborazioni (come ad esempio per gli algoritmi di Knuth, Morris e Pratt e di Boyer e Moore) e prima di trattare metodi di matching esatto più complessi vogliamo far notare che abbiamo già gli strumenti necessari per risolvere il problema del matching esatto in tempo $O(m+n)$. Questo approccio fornisce il più semplice algoritmo di matching esatto in tempo lineare.

Sia $S = P\$T$ la stringa ottenuta concatenando il pattern $P$ con il carattere \$ e quindi con il testo $T$ dove \$ è un simbolo che non compare né in $P$ né in $T$. Siccome $P$ ha lunghezza $m$ e $T$ ha lunghezza $n$ ed $m \leq n$, la stringa $S$ ha lunghezza $m+n+1 = O(m+n)$.

Calcoliamo allora $\pi_i^S$ per $i = 2, \ldots, n+m+1$. Siccome \$ non compare né in $P$ né in $T$ il valore di $\pi_i^S$ è sicuramente minore o uguale ad $m$ per ogni valore di $i$.

Ogni valore di $i \geq m+1$ per il quale $\pi_i^S = m$ identifica una occorrenza di $P$ in $T$ che inizia nella posizione $i-(m+1)$ di $T$.

Viceversa se vi è una occorrenza di $P$ nella posizione $j$ di $T$ il valore di $\pi_{j+(m+1)}^S$ deve essere uguale ad $m$.

Siccome i valori di $\pi_i^S$ per $i = 2, \ldots, n+m+1$ si calcolano in tempo

$$O(n+m+1) = O(m+n)$$

possiamo trovare in tempo lineare tutte le occorrenze di $P$ in $T$ determinando semplicemente tutte le posizioni $i$ maggiori di $m+1$ tali che $\pi_i^S = m$.

Il metodo si può programmare usando soltanto una quantità $O(m)$ di memoria oltre a quella richiesta per memorizzare le due stringhe.

Infatti, siccome $\pi_i^S \leq m$ per ogni posizione $i$, la posizione $k'$ determinata nel Caso 2 dell'algoritmo FUNZIONE-PREFISSO si trova sempre in $P$. Non vi è quindi ragione di memorizzare i valori di $\pi_i^S$ per $i > m$.

Basta quindi memorizzare i valori di $\pi_i^S$ relativi ai caratteri del pattern e mantenere aggiornati i valori di $r$ e di $l$. Questi valori sono sufficienti per calcolare (senza memorizzare) i valori di $\pi_i^S$ per i caratteri del testo e quindi identificare e stampare le posizioni $j$ nel testo in cui $\pi_{j+(m+1)}^S = m$.

Un'altra caratteristica di questo metodo è che esso è un metodo lineare *indipendente dall'alfabeto*. In altre parole non abbiamo bisogno di assumere che l'alfabeto sia noto a priori — sappiamo soltanto che c'è una operazione di confronto tra caratteri che risponde uguale o diverso e questo è tutto quello che ci serve sapere dell'alfabeto.

Vedremo che questo sarà vero anche per gli algoritmi di Knuth, Morris e Pratt e di Boyer e Moore ma non per altri algoritmi come l'algoritmo di Aho e Corasick e quelli basati sull'albero dei suffissi.

## Esercizi

**Esercizio 1** Date due stringhe $X$ ed $Y$ di uguale lunghezza $n$ determinare, in tempo lineare $O(n)$, se $Y$ è una rotazione circolare di $X$, ossia se $Y$ è costituita da un suffisso di $X$ seguito dal prefisso di $X$ di lunghezza complementare.

**Esercizio 2** Date due stringhe $X$ ed $Y$ di lunghezze $m$ ed $n$ calcolare, in tempo lineare $O(m+n)$, il più lungo suffisso di $X$ che è anche prefisso di $Y$.

**Esercizio 3** Nelle sequenze biologiche è importante riconoscere eventuali ripetizioni consecutive della stessa sottostringa. Una tale sequenza di ripetizioni di una sottostringa $A$ in una stringa $T$ viene detta *tandem array* di $A$.

Ad esempio la stringa

$$T = \texttt{cdabcabcabcabcxga}$$

contiene il tandem array `abcabcabcabc` di $A =$ `abc` di lunghezza 4.

Un tandem array è massimale se non può essere esteso né a sinistra né a destra.

Data la *base* $A$ un tandem array di $A$ in $T$ può essere individuato da una coppia di numeri $(s, k)$ in cui $s$ indica la posizione nel testo $T$ dove inizia il tandem array e $k$ è il numero di ripetizioni di $A$.

Mostrare con un esempio che due tandem array massimali di $A$ in un testo $T$ possono sovrapporsi per qualche carattere.

Descrivere un algoritmo che dato un testo $T$ ed una stringa $A$ calcola, in tempo lineare, tutti i tandem array di $A$ massimali fornendo in output le corrispondenti coppie $(s, k)$.

**Esercizio 4** Se l'algoritmo FUNZIONE-PREFISSO, con il primo ciclo while, trova che $\pi_2 = q > 0$ tutti i valori $\pi_3, \pi_4, \ldots, \pi_{q+2}$ si possono calcolare direttamente senza ulteriori confronti tra caratteri. Spiegare come e perché.

**Esercizio 5** In FUNZIONE-PREFISSO il Caso 2b viene eseguito quando $\pi_{k-\ell+1} \geq r-k+1$ ed in tal caso esso confronta i caratteri del pattern con quelli del testo partendo dalla posizione $r - \ell + 2$ del pattern.

Mostrare che se vale la diseguaglianza stretta $\pi_{k-\ell+1} > r - k + 1$ possiamo calcolare direttamente $\pi_k$ senza ulteriori confronti tra caratteri.

**Esercizio 6** Se nell'algoritmo FUNZIONE-PREFISSO, tenendo conto dell'esercizio precedente, dividiamo il Caso 2b in due sottocasi, uno quando $\pi_{k-\ell+1} > r - k + 1$ e l'altro quando $\pi_{k-\ell+1} = r - k + 1$ otteniamo effettivamente un miglioramento nell'efficienza dell'algoritmo?

Discutere questo punto tenendo conto di tutte le operazioni e non soltanto dei confronti tra caratteri.

**Esercizio 7** Uno modo comunemente usato dagli studenti per mascherare una esercitazione C++ copiata è quello di cambiare gli identificatori con altri nomi.

La fase di analisi lessicale di un programma C++ trasforma il programma in una sequenza di *token* (i simboli e le parole riservate del C++) e di *identificatori*.

Quindi, dopo la fase di analisi lessicale, un programma $P$ può essere visto come una stringa di caratteri presi da due alfabeti distinti: un alfabeto $\Sigma$ di token (uguale per tutti i programmi) ed un alfabeto $\Pi_P$ di parametri (gli identificatori del programma).

Diciamo che due programmi $P$ e $Q$ sono *uguali* se, dopo l'analisi lessicale le stringhe che otteniamo soddisfano le seguenti condizioni:

1. Ad ogni token di $P$ corrisponde un token identico in $Q$;
2. Ad ogni parametro di $P$ corrisponde un parametro (non necessariamente uguale) in $Q$;
3. Se un parametro $x$ in $P$ è allineato ad un parametro $y$ in $Q$ allora ogni altra occorrenza di $x$ in $P$ deve essere allineata con una occorrenza di $y$ in $Q$ ed ogni altra occorrenza di $y$ in $Q$ deve essere allineata con una occorrenza di $x$ in $P$.

Ad esempio, se indichiamo con caratteri minuscoli i token e con caratteri maiuscoli i parametri, le stringhe

$$\begin{aligned} P &= \texttt{abVWcVcadYYWe} \\ Q &= \texttt{abYZcYcadXXZe} \end{aligned}$$

sono uguali.

Trovare un algoritmo efficiente per determinare se due programmi sono uguali.

# 3 Metodi classici basati sul confronto di caratteri

## L'algoritmo di Knuth, Morris e Pratt

Come l'algoritmo ingenuo anche l'algoritmo di Knuth, Morris e Pratt [5] allinea il pattern $P$ in posizioni successive del testo $T$ e controlla quindi se i caratteri corrispondenti del pattern e del testo sono uguali. Quando ha terminato questo controllo sposta il pattern alla destra esattamente come nell'algoritmo ingenuo.

L'algoritmo di Knuth, Morris e Pratt tiene però conto di quelle idee che avevamo a suo tempo esposto e che permettono di spostare il pattern per più di una posizione e di ricominciare i confronti non dall'inizio del pattern ma da un carattere successivo.

Supponiamo che con il pattern allineato in posizione $i$ del testo l'algoritmo abbia trovato il primo mismatch confrontando il carattere $P[j]$ con il carattere $T[i+j-1]$ per cui sappiamo che $P[1, j-1] = T[i, i+j-2]$ e che $P[j] \neq T[i+j-1]$. La presenza della sentinella ci assicura che troveremo senz'altro un mismatch per qualche $j \leq m+1$.

Supponiamo vi sia una occorrenza

$$P[1, m] = T[i+h, i+h+m-1]$$

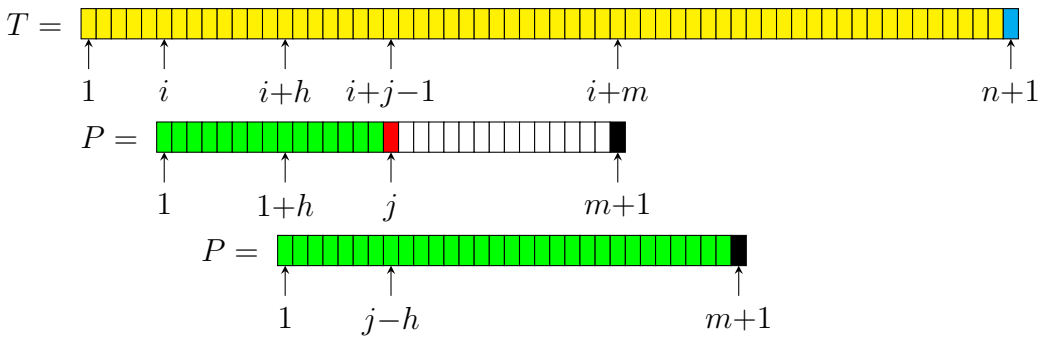



Figura 9: Illustrazione dello spostamento del pattern nell'algoritmo di Knuth, Morris e Pratt. Con il pattern allineato con la posizione $i$ del testo si è trovato il primo mismatch in posizione $j$ (i match sono colorati in verde e i mismatch in rosso) e vi è una occorrenza quando il pattern viene spostato in avanti di una quantità $h$ compresa tra 1 e $j-1$.

quando il pattern viene spostato in avanti di una quantità $h$ compresa tra 1 e $j-1$. In questo caso

$$P[1, j-h] = T[i+h, i+j-1]$$

Siccome $P[1, j-1] = T[i, i+j-2]$ dovrà essere

$$P[1, j-h-1] = T[i+h, i+j-2] = P[1+h, j-1]$$

e siccome $P[j] \neq T[i+j-1]$ dovrà essere pure

$$P[j-h] = T[i+j-1] \neq P[j]$$

La situazione è illustrata in Figura 9.

Dire che $P[1, j-h-1] = P[1+h, j-1]$ e che $P[j-h] \neq P[j]$ è come dire che $\pi^P_{1+h} = j-h-1$ ovvero che

$$j = h + \pi^P_{1+h} + 1$$

Pertanto gli spostamenti $h$ compresi tra 1 e $j-1$ per i quali vi può essere una occorrenza del pattern in posizione $i+h$ del testo sono soltanto quelli per cui $j = h + \pi^P_{1+h} + 1$.

Se questa condizione non è soddisfatta per nessun spostamento $h$ compreso tra 1 e $j-1$ la prima posizione del testo in cui può esserci una occorrenza del pattern è $i+j$.

Pertanto, lo spostamento $d[j]$ del pattern che si deve effettuare quando si è trovato il primo mismatch confrontando $P[j]$ con $T[i+j-1]$ è

$$d[j] = \min(\{j\} \cup \{h \mid 1 \leq h < j,\ j = 1 + h + \pi^P_{1+h}\})$$

Se assumiamo che al pattern sia stata aggiunta la sentinella allora la stessa definizione di $d[j]$ vale anche nel caso in cui sia stata trovata una occorrenza del pattern in posizione $i$ e quindi il mismatch sia stato trovato in corrispondenza della sentinella per $j = m+1$. In questo caso infatti, per $h = m$ la condizione $j = 1 + h + \pi^P_{1+h}$ diventa $m+1 = 1+m+0$

che è sempre vera. Quindi vi è sempre una posizione del testo compresa tra $i+1$ ed $i+m$ in cui può esserci una occorrenza del pattern per cui

$$d[m+1] = \min(\{m+1\} \cup \{h \mid 1 \leq h < m+1,\ m+1 = 1+h+\pi^P_{1+h}\}) \leq m$$

e dunque la definizione precedente di $d[j]$ funziona anche in questo caso.

Il calcolo di $d[j]$ per $j = 1, \ldots, m+1$ si effettua nel modo seguente.

**for** $j = 1$ **to** $m+1$
    // Diamo a ogni $d[j]$ il valore $j$ che è il massimo dei valori tra cui minimizzare.
    $d[j] = j$
**for** $h = m$ **downto** 1
    // Aggiorniamo $d[j]$ quando troviamo un $h$ minore tale che $j = 1+h+\pi_{1+h}$.
    $d[1+h+pref[1+h]] = h$

Dopo aver spostato il pattern di $d[j]$ posti in posizione $i' = i + d[j]$ del testo, se lo spostamento $d[j]$ è minore di $j$ sappiamo che

$$P[1, j-d[j]-1] = T[i+d[j], i+j-2]$$

e possiamo quindi cominciare a confrontare i caratteri del pattern con i corrispondenti caratteri del testo partendo dalla posizione $j' = j - d[j]$ del pattern.

Se invece $d[j] = j$ allora dobbiamo cominciare da $j' = 1$.

L'algoritmo di Knuth, Morris e Pratt è il seguente:

KNUTH-MORRIS-PRATT$(P, T)$
    // $P$ stringa di lunghezza $m$ allungata con una sentinella in posizione $m+1$.
    // $T$ stringa di lunghezza $n \geq m$ allungata anch'essa in posizione $n+1$ con un
    // carattere qualsiasi diverso dalla sentinella.
1  $pref$ = FUNZIONE-PREFISSO$(P)$
2  **for** $j = 1$ **to** $m+1$ **do** $d[j] = j$
3  **for** $h = m$ **downto** 1
4      $d[1+h+pref[1+h]] = h$
5  $i = j = 1$
6  **while** $i \leq n-m+1$
7      **while** $P[j]$ == $T[i+j-1]$
8          $j = j+1$
9      **if** $j > m$
10         "segnala l'occorrenza in posizione $i$"
11     $i = i + d[j]$, $j = \max(1, j-d[j])$

La preelaborazione è costituita dal calcolo della funzione prefisso di $P$ e dai due cicli **for** e richiede tempo $O(m)$.

Il tempo richiesto dalla fase di ricerca è proporzionale al numero di confronti tra caratteri che vengono eseguiti come test del ciclo **while** interno.

Il pattern viene spostato in avanti non appena si incontra un mismatch. Quindi vi è un confronto con esito negativo per ogni posizione del testo in cui viene allineato il pattern.

Siccome tali posizioni sono al massimo $n - m + 1$ vengono eseguiti al più $n - m + 1$ confronti con esito negativo.

Il carattere del testo che viene confrontato è quello in posizione $i+j-1$ e tale carattere si sposta a destra di una posizione ogni volta che viene eseguito un confronto con esito positivo. Siccome $i + j - 1$ ha valore 1 quando inizia la fase di ricerca e quando il pattern viene spostato in avanti o rimane invariato o aumenta di 1 (la diminuzione di $j$ è minore o uguale all'aumento di $i$) e alla fine $i+j-1$ è minore o uguale ad $n+1$, possiamo concludere che vengono eseguiti al più $n$ confronti con esito positivo (quando $i + j - 1 = n + 1$ c'è senz'altro un mismatch).

La complessità dell'algoritmo di Knuth, Morris e Pratt è quindi $O(m + n)$.

## L'algoritmo di Boyer e Moore

L'algoritmo di Boyer e Moore [3], a differenza di quello di Knuth, Morris e Pratt, confronta i caratteri del pattern con quelli del testo nell'ordine *da destra a sinistra*. Osserviamo che in questo caso dobbiamo aggiungere la sentinella all'inizio del pattern in posizione $P[0]$ e allungare il testo con un primo carattere $T[0]$ diverso dalla sentinella.

Supponiamo che con la sentinella $P[0]$ allineata con il carattere $T[i]$ l'algoritmo abbia trovato il primo mismatch confrontando il carattere $P[j]$ con il carattere $T[i+j]$. Siccome l'algoritmo confronta i caratteri da destra a sinistra sappiamo a questo punto che

$$P[j+1, m] = T[i+j+1, i+m]$$

e che $P[j] \neq T[i+j]$.

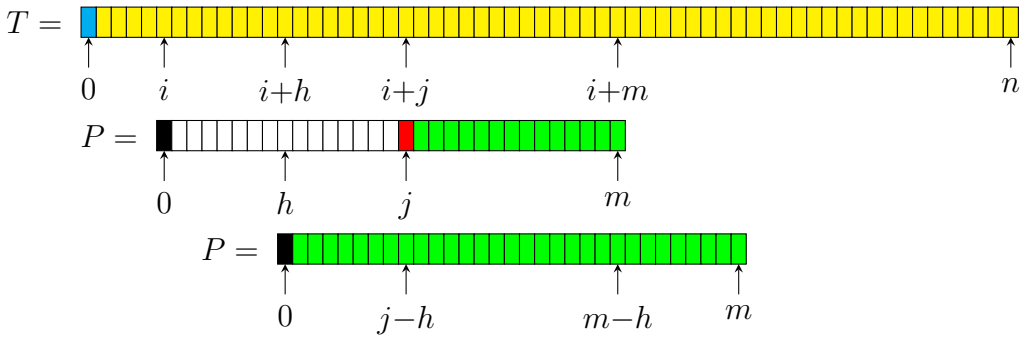



Figura 10: Con la sentinella $P[0]$ allineata con il carattere $T[i]$ e partendo dal fondo si è trovato il primo mismatch in posizione $j$ (i match sono colorati in verde e i mismatch in rosso) e vi è una occorrenza del pattern quando la sentinella $P[0]$ è allineata con $T[i + h]$ per uno spostamento $h$ compreso tra 1 e $j - 1$.

Supponiamo vi sia una occorrenza

$$P[1, m] = T[i + h + 1, i + h + m]$$

del pattern quando la sentinella $P[0]$ è allineata con $T[i + h]$ per uno spostamento $h$ compreso tra 1 e $j - 1$.

In questo caso deve essere

$$P[j-h, m-h] = T[i+j, i+m]$$

Siccome $P[j+1, m] = T[i+j+1, i+m]$ dovrà essere

$$P[j-h+1, m-h] = T[i+j+1, i+m] = P[j+1, m]$$

e siccome $P[j] \neq T[i+j]$ dovrà essere

$$P[j-h] = T[i+j] \neq P[j]$$

La situazione è illustrata in Figura 10.

Sia $P_{rev}$ la stringa ottenuta ribaltando il pattern $P$, ossia se $P = x_0 x_1 \dots x_m$ con $x_0$ sentinella allora $P_{rev} = y_1 \dots y_m y_{m+1} = x_m \dots x_1 x_0$. Quindi $P_{rev}[m-i+1] = P[i]$ per ogni $i$ e $P_{rev}[m+1] = P[0]$ è la sentinella.

Dire che $P[j-h+1, m-h] = P[j+1, m]$ e che $P[j-h] \neq P[j]$ è equivalente a dire che

$$P_{rev}[1+h, m-j+h] = P_{rev}[1, m-j]$$

e che

$$P_{rev}[m-j+h+1] \neq P_{rev}[m-j+1]$$

Questo è come dire che $\pi^{P_{rev}}_{1+h} = m-j$ ovvero che

$$j = m - \pi^{P_{rev}}_{1+h}$$

Pertanto, gli spostamenti $h$ compresi tra 1 e $j-1$ tali che ci possa essere una occorrenza del pattern quando la sentinella $P[0]$ è allineata con $T[i+h]$ sono soltanto quelli per cui $j = m - \pi^{P_{rev}}_{1+h}$.

Supponiamo ora che vi sia una occorrenza

$$P[1, m] = T[i+h+1, i+h+m]$$

quando la sentinella $P[0]$ è allineata con il carattere $T[i+h]$ per uno spostamento $h$ compreso tra $j$ ed $m$.

In questo caso deve essere

$$P[1, m-h] = T[i+h+1, i+m]$$

e siccome $P[j+1, m] = T[i+j+1, i+m]$ dovrà essere

$$P[h+1, m] = P[1, m-h]$$

La situazione è illustrata in Figura 11.

Dire che $P[h+1, m] = P[1, m-h]$ è come dire che il pattern $P$, e quindi anche $P_{rev}$, ha un bordo di lunghezza $m-h$ e questo a sua volta è equivalente a dire che $\pi^{P_{rev}}_{h+1} = m-h$.

Pertanto, gli spostamenti $h$ compresi tra $j$ ed $m$ tali che ci possa essere una occorrenza del pattern quando la sentinella $P[0]$ è allineata con $T[i+h]$ sono soltanto quelli per cui $\pi^{P_{rev}}_{h+1} = m-h$.

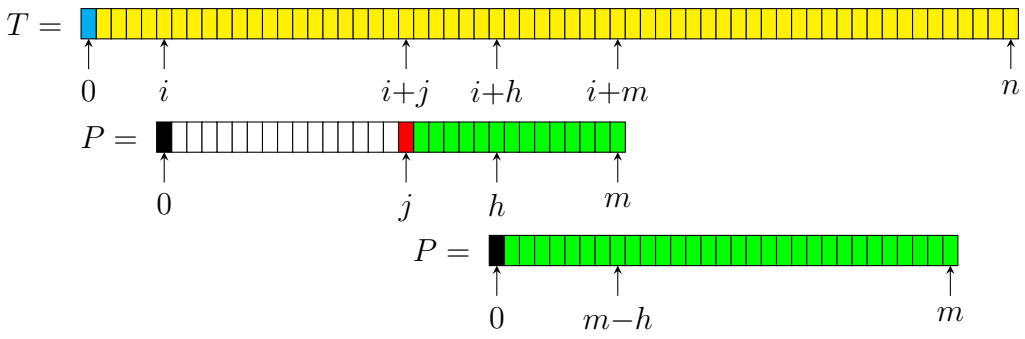



Figura 11: Con la sentinella $P[0]$ allineata con il carattere $T[i]$ e partendo dal fondo si è trovato il primo mismatch in posizione $j$ (i match sono colorati in verde e i mismatch in rosso) e vi è una occorrenza del pattern quando la sentinella $P[0]$ è allineata con $T[i+h]$ per uno spostamento $h$ compreso tra $j$ ed $m$.

Osserviamo che per $h = m$ abbiamo $\pi^{P_{rev}}_{1+m} = 0$ poiché in posizione $m+1$ di $P_{rev}$ vi è la sentinella. Di conseguenza vi è sempre almeno un valore di $h$ per il quale $\pi^{P_{rev}}_{h+1} = m - h$. Inoltre, per $h = 0$ abbiamo $\pi^{P_{rev}}_{1+0} = m + 1 \neq m + 0$ e quindi $h = 0$ non soddisfa la condizione.

Quindi, lo spostamento $d[j]$ del pattern che si deve effettuare quando, con la sentinella $P[0]$ allineata con il carattere $T[i]$ del testo e partendo dalla fine del pattern, si sia trovato il primo mismatch confrontando il carattere $P[j]$ con il carattere $T[i+j]$ è

$$\begin{aligned} d[j] \;=\; & \min(\{h \mid 1 \leq h < j,\ j = m - \pi^{P_{rev}}_{1+h}\} \cup \\ & \{h \mid j \leq h \leq m,\ h = m - \pi^{P_{rev}}_{1+h}\}\ ) \end{aligned}$$

Il calcolo di $d[j]$ per $j = 0, \ldots, m$ si effettua nel modo seguente.

// Calcoliamo dapprima $d[j] = \min(\{h \mid j \leq h \leq m,\ h = m - \pi^{P_{rev}}_{1+h}\})$
**for** $j = m$ **downto** $0$
    **if** $j$ == $m - pref[1+j]$
        $h = j$
    // $h$ è il mimimo tale che $\max(1, j) \leq h \leq m$ ed $h = m - \pi^{P_{rev}}_{1+h}$
    $d[j] = h$
// Se $\{h \mid 1 \leq h < j,\ j = m - \pi^{P_{rev}}_{1+h}\}$ non è vuoto poniamo
// $d[j] = \min(\{h \mid 1 \leq h < j,\ j = m - \pi^{P_{rev}}_{1+h}\})$
**for** $h = m$ **downto** $1$
    $j = m - pref[1+h]$
    **if** $h < j$
        $d[j] = h$

dove $pref[j] = \pi^{P_{rev}}_{j}$ è la funzione prefisso calcolata per la stringa $P_{rev}$. L'algoritmo di Boyer e Moore è quindi il seguente.

```
BOYER-MOORE(P, T)
    // P stringa di lunghezza m con l'aggiunta di una sentinella in posizione 0.
    // T stringa di lunghezza n ≥ m anch'essa con l'aggiunta in posizione 0 di un
    // carattere qualsiasi diverso dalla sentinella.
 1  pref = FUNZIONE-PREFISSO(P_rev)
 2  for j = m downto 0
 3      if j == m − pref[1 + j]
 4          h = j
 5      d[j] = h
 6  for h = m downto 1
 7      j = m − pref[1 + h]
 8      if h < j
 9          d[j] = h
10  i = 0
11  while i ≤ n − m
12      j = m
13      while P[j] == T[i + j]
14          j = j − 1
15      if j == 0
16          "segnala l'occorrenza in posizione i + 1"
17      i = i + d[j]
```

La preelaborazione è costituita dal calcolo della funzione prefisso di $P_{rev}$ e dai due cicli **for** e richiede tempo $O(m)$.

La dimostrazione che il tempo richiesto dalla fase di ricerca è lineare nella lunghezza del testo è estremamente complessa e pertanto la ometteremo.

Ci limitiamo qui ad osservare che a differenza dell'algoritmo di Knuth, Morris e Pratt in cui lo spostamento dal pattern non è mai superiore al numero di caratteri confrontati nella posizione attuale del pattern, nell'algoritmo di Boyer e Moore lo spostamento può essere maggiore del numero di confronti effettuati in tale posizione.

In effetti nella maggior parte dei casi, e soprattutto per pattern lunghi, l'algoritmo di Boyer e Moore richiede un numero totale di confronti che è soltanto una frazione del numero di caratteri del testo.

Questo lo rende preferibile nei casi pratici, anche se nel caso pessimo il numero di confronti richiesto è $4n$, doppio rispetto a Knuth, Morris e Pratt.

## L'algoritmo di Knuth, Morris e Pratt on-line

A differenza dell'algoritmo di Knuth, Morris e Pratt semplice, l'algoritmo di Knuth, Morris e Pratt on-line assume che l'alfabeto sia finito e noto a priori.

Supponiamo che con il pattern allineato in posizione $i$ del testo l'algoritmo abbia scoperto che i primi $j-1$ caratteri del pattern sono uguali ai corrispondenti caratteri del testo e sia $k = i + j - 1$ la posizione del testo allineata con la posizione $j$ del pattern per cui

$$P[1, j-1] = T[i, k-1]$$

Supponiamo vi sia una occorrenza del pattern in una posizione $i+h$ del testo compresa tra $i$ e $i+j-1$.

In questo caso

$$P[1, j-h] = T[i+h, k]$$

Siccome $P[1, j-1] = T[i, k-1]$ dovrà essere

$$P[1, j-h-1] = T[i+h, k-1] = P[1+h, j-1]$$

ed inoltre

$$P[j-h] = T[k]$$

La situazione è illustrata in Figura 12.

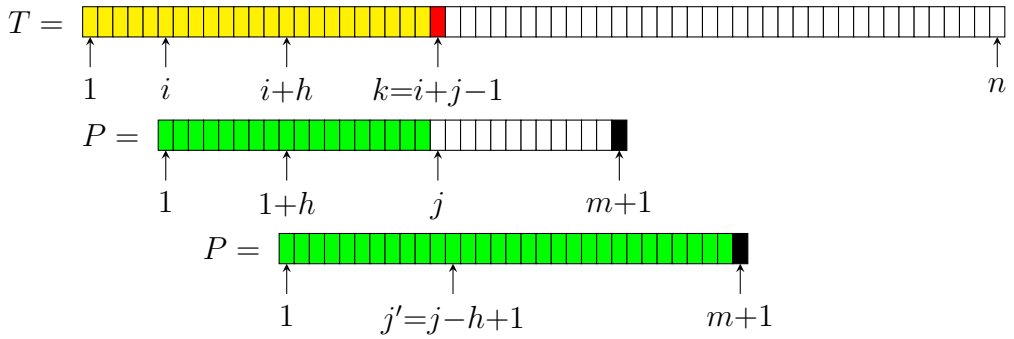



Figura 12: Il pattern allineato con la posizione $i$ del testo ha i primi $j-1$ caratteri uguali ai corrispondenti caratteri del testo. Vi può essere una occorrenza del pattern in una posizione $i+h$ compresa tra $i$ e $i+j-1$ soltanto se $P[1, j-h-1] = P[1+h, j-1]$ e il successivo carattere $T[k]$ del testo è uguale al carattere $P[j-h]$ del pattern.

Pertanto, la prima posizione del testo compresa tra $i$ e $k = i+j-1$ in cui può esserci una occorrenza del pattern è la prima posizione $i+h$ per cui

$$P[1, j-h-1] = P[1+h, j-1] \quad \text{e} \quad P[j-h] = T[k]$$

Se $T[k] = P[j]$ allora la prima posizione è $i+0 = i$.

Se $T[k] \neq P[j]$ allora $i+h$ è la prima posizione per cui

$$P[1, j-h-1] = P[1+h, j-1] \quad \text{e} \quad P[j-h] = T[k] \neq P[j]$$

Questo è come dire che $\pi^P_{1+h} = j-h-1$ e $P[j-h] = T[k]$.

Se tale condizione non è verificata in nessuna posizione compresa tra $i$ ed $i+j-1$ allora la prima posizione del testo in cui può esserci una occorrenza del pattern è $i+j$.

Pertanto lo spostamento in funzione di $j = 1, \ldots, m+1$ e del carattere $x = T[k]$ è uguale a 0 se $P[j] = x$ mentre se $P[j] \neq x$ esso è

$$h(j, x) = \min(\{j\} \cup \{h \mid 0 < h < j,\ j = 1 + h + \pi^P_{1+h},\ P[1 + \pi^P_{1+h}] = T[k]\})$$

Se l'alfabeto è finito e noto a priori possiamo calcolare, in una fase di preelaborazione del pattern, il valore di tale spostamento $h(j, x)$ per ogni valore di $j = 1, \ldots, m + 1$ e per ogni possibile valore di $T[k]$ (ossia per ogni carattere $x$ dell'alfabeto).

Osserviamo che dopo lo spostamento o il pattern è allineato sul carattere successivo a $T[k]$ (se lo spostamento è $j$) oppure $T[k]$ è uguale al corrispondente carattere $P[j - h]$ del pattern. In entrambi i casi possiamo passare al carattere successivo $T[k + 1]$ del testo che risulta allineato con il carattere $P[j - h + 1]$ del pattern.

L'algoritmo di Knuth, Morris e Pratt on-line prende quindi in considerazione ogni carattere del testo una e una sola volta, e per questo viene appunto detto on-line.

Per ogni carattere del testo l'algoritmo sposta il pattern di una quantità $h$ (eventualmente anche $h = 0$) che dipende dal carattere stesso e dalla posizione $j$ del carattere del pattern che è allineato con il carattere del testo considerato.

Siccome l'unica informazione che serve all'algoritmo per il passo successivo è la posizione $j' = j - h + 1$ del carattere del pattern che risulterà allineato con il carattere $T[k+1]$ del testo dopo lo spostamento, assumiamo che il risultato della preelaborazione sia una tabella

$$\delta[j, x] = j - h(j, x) + 1$$

che per ogni $j = 1, \ldots, m + 1$ e per ogni carattere $x$ dell'alfabeto restituisce la posizione $j' = \delta[j, x]$ del carattere del pattern che risulterà allineato con il carattere del testo $T[k+1]$ dopo lo spostamento relativo al caso $T[k] = x$.

Quando $j' = m+1$ avremo che $P[1, m] = T[k-m+1, k]$ e quindi possiamo segnalare una occorrenza del pattern in posizione $k - m + 1$.

Osserviamo che $\delta[j, x]$ è la lunghezza del più lungo suffisso di $P[1, j - 1]x$ che è prefisso di $P$ aumentata di 1.

Ricordando la definizione di $h(j, x)$ abbiamo

$$\delta[j, x] = \begin{cases} j + 1 & \text{se } P[j] = x \\ \max\big(\{1\} \cup \{j - h + 1 \mid 0 < h < j, \\ \qquad j = 1 + h + \pi^P_{1+h}, P[j - h] = x\}\big) & \text{altrimenti} \end{cases}$$

Pertanto $\delta[j, x]$ si può calcolare in tempo lineare nella fase di preelaborazione del pattern nel modo seguente.

**for** $j = 1$ **to** $m + 1$
    **for** "ogni carattere $x$ dell'alfabeto"
        $\delta[j, x] = 1$
**for** $j = 1$ **to** $m$
    $\delta[j, P[j]] = j + 1$
**for** $h = m$ **downto** 1
    $j = 1 + h + pref[1 + h]$
    $\delta[j, P[j - h]] = j - h + 1$

Ad esempio la tabella $\delta$ per il pattern $P =$ `abcdabca` è la seguente.

| $j \backslash x$ | a | b | c | d | $P$ |
|---|---|---|---|---|---|
| 1 | 2 | 1 | 1 | 1 | a |
| 2 | 2 | 3 | 1 | 1 | b |
| 3 | 2 | 1 | 4 | 1 | c |
| 4 | 2 | 1 | 1 | 5 | d |
| 5 | 6 | 1 | 1 | 1 | a |
| 6 | 2 | 7 | 1 | 1 | b |
| 7 | 2 | 1 | 8 | 1 | c |
| 8 | 9 | 1 | 1 | 5 | a |
| 9 | 2 | 3 | 1 | 1 | |

L'algoritmo di Knuth, Morris e Pratt on-line è allora il seguente.

KNUTH-MORRIS-PRATT-ON-LINE$(P, T, \Sigma)$

```
1   // P stringa di lunghezza m allungata con una sentinella in posizione m + 1.
    // T stringa di lunghezza n ≥ m. Σ alfabeto finito.
2   pref = FUNZIONE-PREFISSO(P)
3   for j = 1 to m + 1
4       for "ogni carattere x dell'alfabeto"
5           δ[j, x] = 1
6   for j = 1 to m
7       δ[j, P[j]] = j + 1
8   for h = m downto 1
9       j = 1 + h + pref[1 + h]
10      δ[j, P[j − h]] = j − h + 1
11  j = 1
12  for k = 1 to n
13      j = δ[j, T[k]]
14      if j == m + 1
15          "segnala l'occorrenza in posizione k − m + 1"
```

Con $P =$ `abcdabca` e $T =$ `cabcdabcdabcab`, l'algoritmo calcola dapprima la tabella $\delta$ vista precedentemente e quindi, per $k = 1, \ldots, 14$ calcola i seguenti valori di $j$.

| $T$ | c | a | b | c | d | a | b | c | d | a | b | c | a | b | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $j$ | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 5 | 6 | 7 | 8 | **9** | 3 |
| $P$ | | | | | | a | b | c | d | a | b | c | a | | |

La tavola $\delta$ si può rappresentare con un grafo i cui nodi sono le posizioni $j$ del pattern e per ogni $j' = \delta(j, x)$ un arco $(j, j')$ etichettato con il carattere $x$. Ad esempio, la tavola $\delta$ dell'esempio precedente si rappresenta con il grafo di Figura 13.

Siccome ad ogni posizione $j$ corrisponde un prefisso del pattern possiamo anche dire che i nodi rappresentano i prefissi del pattern e che vi è un arco etichettato con il carattere $x$ che connette un nodo che rappresenta un prefisso $\alpha$ del pattern con un nodo che rappresenta un prefisso $\beta$ del pattern esattamente quando $\beta$ è il più lungo prefisso del pattern che è anche un suffisso della stringa $\alpha x$, ossia $\beta = \alpha x$ se $\alpha x$ è un prefisso di $P$ altrimenti $\beta$ è il più lungo bordo proprio di $\alpha x$.

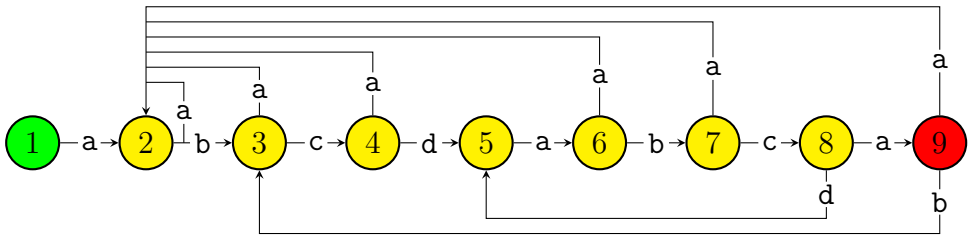



Figura 13: Il grafo di transizione dell'algoritmo di Knuth, Morris e Pratt on line per il pattern $P = \texttt{abcdabca}$. Per non appesantire la figura gli archi che portano al primo nodo sono stati omessi. Pertanto se non c'è un arco uscente da un nodo $j$ con etichetta il carattere x è sottointeso che vi è un arco con etichetta x che connette $j$ al nodo 1. Il nodo verde 1 è il nodo iniziale. Si arriva al nodo rosso 9 quando si è trovata una occorrenza del pattern.

## L'algoritmo di Aho e Corasick

L'algoritmo di Aho e Corasick [1] risolve il problema del matching esatto per un insieme $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ di pattern. Esso calcola tutte le occorrenze dei pattern in $\mathcal{P}$ in un testo $T$ di lunghezza $n$ in tempo lineare.

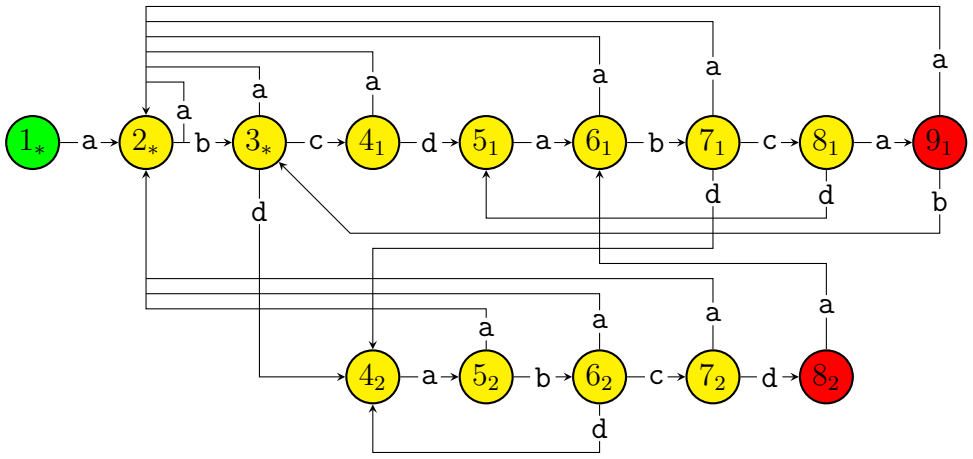



Figura 14: Il grafo di transizione dell'algoritmo di Aho e Corasick per i due pattern $P_1 = \texttt{abcdabca}$ e $P_2 = \texttt{abdabcd}$. I nodi $1_*$, $2_*$ e $3_*$ sono in comune in quanto i due pattern hanno il prefisso ab in comune. Il nodo verde $1_*$ è il nodo iniziale. Si arriva ai nodi rossi $9_1$ e $8_2$ quando si è trovata una occorrenza del corrispondente pattern $P_1$ o $P_2$.

L'algoritmo di Aho e Corasick è una generalizzazione dell'algoritmo di Knuth, Morris e Pratt on-line.

Nella fase di preelaborazione viene costruito un grafo di transizione avente un nodo per ogni prefisso distinto dei pattern $P_1, P_2, \ldots, P_k$ ed in cui vi è un arco etichettato con il carattere $x$ che congiunge il prefisso $\alpha$ al prefisso $\beta$ se $\beta$ è il più lungo suffisso di $\alpha x$ che è prefisso di uno dei pattern $P_1, P_2, \ldots, P_k$.

Nella fase di ricerca viene segnalata una occorrenza del pattern $P_i$ ogni volta che si arriva nel nodo relativo ad un prefisso $\alpha = P_i$ uguale ad uno dei pattern cercati.

Ad esempio il grafo di transizione relativo ai due pattern `abcdabca` ed `abdabcd` è quello illustrato in Figura 14.

La preelaborazione per il metodo di Aho e Corasick si compone dei seguenti passi;

1. calcolare per ogni $s$ e $t$ la funzione $\pi_j^{s,t}$: la lunghezza del più lungo prefisso di $P_s$ che compare in posizione $j$ di $P_t$. Questa si puòcalcolare calcolando dapprima la funzione prefisso $\pi_j^{P_s\$P_t}$ della stringa $P_s\$P_t$ e ponendo quindi $\pi_j^{s,t} = \pi_{j+m_s+1}^{P_s\$P_t}$.

2. calcolare per ogni $s$ e $t$

$$\delta^{s,t}[j,x] = \begin{cases} j+1 & \text{se } P_s[j] = x \\ \max\left(\{1\} \cup \{j-h+1 \,|\, 0 < h < j, \right. & \\ \quad \left. j = 1 + h + \pi_{1+h}^{s,t}, P_s[j-h] = x\}\right) & \text{altrimenti} \end{cases}$$

3. calcolare per ogni $t$

$$\delta[j,t,x] = \max_{s=1,\ldots,k} \left(\{(j',s) \mid j' = \delta^{s,t}[j,x]\}\right)$$

Se $j' = \delta^{s',t}[j,x] = \delta^{s'',t}[j,x]$ prendiamo $s = \min(s',s'')$.

**Esercizio 8** Se l'alfabeto è finito e noto a priori anche l'algoritmo di Boyer e Moore può essere modificato tenendo conto del carattere del testo $T[k]$ su cui si è verificato il mismatch.

Infatti se $T[k] = x$ e $P[\ell_x, m]$ è il più lungo suffisso del pattern $P$ che non contiene il carattere $x$, allora non ci può essere alcuna occorrenza del pattern nelle posizioni in cui il carattere $T[k]$ del testo risulta allineato con uno dei caratteri del suffisso $P[\ell_x, m]$.

Quindi, se il mismatch si è avuto confrontando $T[k] = x$ con il carattere $P[j]$ del pattern, lo spostamento del pattern deve essere almeno pari $j - \ell_x + 1$.

Se l'alfabeto è finito e noto a priori possiamo calcolare i valori di $\ell_x$ per ogni carattere $x$ dell'alfabeto nella fase di preelaborazione del pattern e quindi definire lo spostamento come

$$\begin{aligned} d[j] \;=\; & \min(\{h \mid \max(1, j-\ell_x+1) \le h < j,\ j = m - \pi_{1+h}^{P_{rev}}\} \cup \\ & \{h \mid j \le h \le m,\ h = m - \pi_{1+h}^{P_{rev}}\}\ ) \end{aligned}$$

Scrivere l'algoritmo di Boyer e Moore così modificato.

**Esercizio 9** Supponiamo che i caratteri del pattern $P$ e del testo $T$ siano scelti casualmente ed indipendentemente in un alfabeto finito.

Trovare un limite superiore al valore atteso del numero di confronti richiesto dall'algoritmo dell'esercizio precedente in funzione della lunghezza $n$ del testo, della lunghezza $m$ del pattern e del numero di caratteri $k$ dell'alfabeto.

# 4 Metodi seminumerici

Tutti gli algoritmi visti finora si basano sul confronto tra caratteri. Ci sono anche algoritmi di string matching che usano operazioni su bit oppure operazioni aritmetiche invece dei confronti tra caratteri. In questa sezione vediamo i due principali metodi di questo tipo.

## Il metodo SHIFT-AND

Il metodo è dovuto ad R. Baeza-Yates e G. Gonnet [2] e funziona molto bene per pattern relativamente corti (ad esempio della lunghezza di una normale parola di una lingua parlata).

Gli autori lo hanno chiamato SHIFT-OR ma SHIFT-AND sembra più naturale ed è con quest'ultimo nome che oggi è noto.

Sia $P$ un pattern di lunghezza $m$ e $T$ un testo di lunghezza $n > m$. Il metodo SHIFT-AND calcola successivamente le righe $M_0[1, m], \ldots, M_n[1, m]$ di una matrice booleana $M$ di dimensioni $(n + 1) \times m$ definita nel modo seguente

**Definizione 4.1** *$M_i[j] = 1$ se e solo se il prefisso $P[1, j]$ di lunghezza $j$ occorre nel testo in posizione $i - j + 1$ (e quindi termina in posizione $i$), ossia se e solo se*

$$P[1, j] = T[i - j + 1, i]$$

*Altrimenti $M_i[j] = 0$.*

Da questa definizione segue che $M_i[m] = 1$ se e solo se vi è una occorrenza del pattern $P$ nella posizione $i - m + 1$ del testo $T$. Dunque l'ultima colonna di $M$ contiene la soluzione del problema del pattern matching esatto.

L'algoritmo prima di costruire $M$, che è la fase di ricerca, effettua una fase di preelaborazione del pattern in cui costruisce, per ogni carattere $x$ dell'alfabeto, un vettore booleano $U_x$ di lunghezza $m$ ponendo $U_x[j] = 1$ se $P[j] = x$, $U_x[j] = 0$ altrimenti. In altre parole $U_x$ memorizza le posizioni nel pattern in cui compare il carattere $x$. L'algoritmo usa la seguente operazione BIT-SHIFT sui vettori booleani

**Definizione 4.2** *Sia $R$ un vettore booleano di lunghezza $m$.* BIT-SHIFT$(R)$ *è il vettore booleano che si ottiene da $R$ spostando tutti i bit di una posizione verso destra. L'ultimo bit a destra viene perso mentre a sinistra entra un bit 1.*

Ad esempio,

$$\begin{aligned} R &= 0111001000101 \\ \text{BIT-SHIFT}(R) &= 1011100100010 \\ \text{BIT-SHIFT}(\text{BIT-SHIFT}(R)) &= 1101110010001 \\ &\textit{ecc.} \end{aligned}$$

La maggior parte dei processori prevede una istruzione macchina che effettua tale operazione sui bit di una parola di memoria (usualmente detta shift a destra o shift aritmetico).

Quindi, se la lunghezza del pattern $m$ è minore o uguale alla lunghezza di una parola di memoria, l'operazione BIT-SHIFT$(R)$ si esegue con una singola istruzione macchina (anche se occupa due o più parole la si può generalmente realizzare con poche istruzioni macchina).

La costruzione di $M$ si effettua riga per riga.

La prima riga $M_0$ ha tutti i bit uguali a 0 poiché nessun prefisso non nullo del pattern può occorrere nel testo terminando in posizione 0 (ossia prima che inizi il testo).

Dopo di che la riga $i$-esima si ottiene dalla riga $(i-1)$-esima e dal vettore $U_{T[i]}$ nel modo seguente:

$$M_i = \text{AND}(\text{BIT-SHIFT}(M_{i-1}), U_{T[i]})$$

dove AND è l'operazione booleana "and" bit a bit: un'altra operazione che si può normalmente eseguire con una sola istruzione macchina su qualsiasi processore.

L'algoritmo completo SHIFT-AND è quindi il seguente (dove $[0, 0, \ldots, 0]$ è il vettore booleano nullo: quello costituito da $m$ bit uguali a 0).

SHIFT-AND$(P, T, \Sigma)$

```
1  // P stringa di lunghezza m. T stringa di lunghezza n ≥ m. Σ alfabeto finito.
2  for "ogni carattere x dell'alfabeto Σ"
3      U_x = [0, 0, ..., 0]
4  for j = 1 to m
5      U_{P[j]}[j] = 1
6  M_0 = [0, 0, ..., 0]
7  for i = 1 to n
8      M_i = AND(BIT-SHIFT(M_{i-1}), U_{T[i]})
9      if M_i[m] = 1
10         "segnala l'occorrenza in posizione i − m + 1"
```

**Teorema 4.3** *L'algoritmo* SHIFT-AND *calcola correttamente tutte le occorrenze del pattern $P$ nel testo $T$.*

**Dimostrazione.** La prima riga $M_0$ viene correttamente calcolata azzerando tutti i bit.

Assumiamo quindi induttivamente che la riga $M_{i-1}$ sia stata calcolata correttamente e mostriamo che l'algoritmo calcola correttamente la riga $M_i$.

Ogni bit $M_i[j]$ viene calcolato come "and" booleana dei due bit in posizione $j$ nei vettori BIT-SHIFT$(M_{i-1})$ ed $U_{T[i]}$.

Il primo bit di BIT-SHIFT$(M_{i-1})$ è 1 e quindi $M_i[1] = U_{T[i]}[1]$ è 1 se e solo se $P[1] = T[i]$, ossia se e solo se vi è una occorrenza del prefisso $P[1, 1]$ di lunghezza 1 in posizione $i - 1 + 1 = i$ nel testo. Quindi il primo bit della riga $i$-esima è calcolato correttamente.

Il bit in posizione $j \geq 2$ di BIT-SHIFT$(M_{i-1})$ è uguale al bit $M_{i-1}[j-1]$ in posizione $j-1$ nella riga precedente e dunque esso è 1 se e solo se

$$P[1, j-1] = T[i-j+1, i-1]$$

Siccome $U_{T[i]}[j] = 1$ se e solo se $P[j] = T[i]$ la "and" booleana tra il bit $j$-esimo di BIT-SHIFT$(M_{i-1})$ ed il bit $j$-esimo di $U_{T[i]}$ ha risultato 1 se e solo se

$$P[1, j] = T[i-j+1, i]$$

Dunque anche i bit successivi della riga $i$-esima sono calcolati correttamente. □

Ad esempio, con il pattern $P = \texttt{abaca}$ ed il testo $T = \texttt{ababacabacac}$ l'algoritmo calcola dapprima

$$\begin{aligned} U_{\texttt{a}} &= 10101 \\ U_{\texttt{b}} &= 01000 \\ U_{\texttt{c}} &= 00010 \end{aligned}$$

e quindi calcola le righe della matrice $M$

$$\begin{aligned} M_1 &= 10000 \\ M_2 &= 01000 \\ M_3 &= 10100 \\ M_4 &= 01000 \\ M_5 &= 10100 \\ M_6 &= 00010 \\ M_7 &= 10001 \\ M_8 &= 01000 \\ M_9 &= 10100 \\ M_{10} &= 00010 \\ M_{11} &= 10001 \\ M_{12} &= 00000 \end{aligned}$$

e segnala correttamente le due occorrenze nelle posizioni $7 - 5 + 1 = 3$ e $11 - 5 + 1 = 7$.

Osserviamo che per il calcolo della riga $i$-esima serve soltanto conoscere la riga precedente $(i-1)$-esima. Non occorre quindi riservare memoria per l'intera matrice $M$. Basta usare un solo array booleano $R$ di lunghezza $m$ per memorizzare la riga che di volta in volta viene calcolata.

Possiamo semplificare quindi l'algoritmo nel seguente modo.

SHIFT-AND$(P, T, \Sigma)$

```
1  // P stringa di lunghezza m. T stringa di lunghezza n ≥ m. Σ alfabeto finito.
2  for "ogni carattere x dell'alfabeto Σ"
3      U_x = [0, 0, ..., 0]
4  for j = 1 to m
5      U_{P[j]}[j] = 1
6  R = [0, 0, ..., 0]
7  for i = 1 to n
8      R = AND(BIT-SHIFT(R), U_{T[i]})
9      if R[m] = 1
10         "segnala l'occorrenza in posizione i − m + 1"
```

Contando le operazioni sui singoli bit l'algoritmo SHIFT-AND ha complessità $\Theta(nm)$.

Se il pattern è più corto della lunghezza di una parola di memoria le operazioni BIT-SHIFT e AND possono essere eseguite contemporaneamente su tutti i bit dei vettori booleani con una singola istruzione macchina richiedendo un tempo costante (e normalmente anche molto piccolo).

In questo caso la complessità si riduce a $\Theta(n)$ con una costante nascosta nella notazione asintotica che è molto piccola. Per questa ragione il metodo SHIFT-AND risulta estremamente efficiente per pattern non troppo lunghi.

## Il metodo dell'impronta di Karp e Rabin

Per semplicità di esposizione esporremo il metodo dell'impronta di Karp e Rabin [4] per il caso in cui pattern e testo siano stringhe sull'alfabeto binario $\{0, 1\}$ in modo che pattern e testo possono essere visti come due numeri interi rappresentati in base 2.

L'estensione al caso generale di un alfabeto di $m > 2$ caratteri si può fare usando come base della numerazione la più piccola potenza $2^k$ maggiore o uguale al numero di caratteri $m$ dell'alfabeto, interpretando i caratteri come cifre in base $2^k$ e quindi le stringhe come numeri interi scritti in base $2^k$ (ad esempio se si usa il codice ASCII la base da usare è $2^8 = 256$). Lasciamo al lettore la facile estensione.

Avendo interpretato le stringhe come numeri interi possiamo utilizzare le operazioni aritmetiche invece dei confronti tra caratteri.

Sia quindi $P$ un pattern binario di lunghezza $m$ e $T$ un testo pure binario di lunghezza $n \geq m$. Cominciamo con alcune definizioni

**Definizione 4.4** *Per $i = 1, \ldots, n - m + 1$, indichiamo con*

$$T_i = T[i, i + m - 1]$$

*la sottostringa di lunghezza $m$ di $T$ che inizia in posizione $i$.*

**Definizione 4.5** *Per ogni stringa binaria $S$ di lunghezza $m$ indichiamo con*

$$H(S) = \sum_{j=1}^{m} 2^{m-j} S[j]$$

*il numero intero la cui rappresentazione binaria è $S$.*

Quindi per il pattern $P$ abbiamo

$$H(P) = \sum_{j=1}^{m} 2^{m-j} P[j]$$

mentre per le sottostringhe $T_i$ del testo

$$H(T_i) = \sum_{j=1}^{m} 2^{m-j} T[i + j - 1]$$

Chiaramente $H(P) = H(T_i)$ quando esiste una occorrenza di $P$ nella posizione $i$ di $T$. Inoltre, siccome ogni intero si può scrivere in modo unico come somma di potenze di 2, è vero anche il contrario e quindi

**Teorema 4.6** *$H(P) = H(T_i)$ se e solo se esiste una occorrenza del pattern $P$ nella posizione $i$ del testo $T$.*

Il teorema precedente permette di trasformare il problema del pattern matching in un problema numerico sostituendo i confronti tra caratteri con confronti tra numeri.

Purtroppo, a meno che il pattern non sia molto corto $H(P)$ e gli $H(T_i)$ non si possono calcolare in modo efficiente. Il problema è che il numero di bit necessari per rappresentare tali numeri interi cresce con la lunghezza $m$ del pattern.

Dal punto di vista della complessità questo contraddice l'ipotesi del modello della macchina con memoria RAM. In tale modello infatti possiamo assumere che le operazioni aritmetiche siano eseguite in tempo costante soltanto se i numeri che si usano sono rappresentabili con al più $O(\log n)$ bit dove $n$ è la dimensione dell'input.

Nel nostro caso la dimensione dell'input è $m + n$ ed il numero di bit per rappresentare gli interi $H(P)$ ed $H(T_i)$ è $m$ e dunque $m = O(\log(m + n))$ soltanto se $m = O(\log n)$, ossia per pattern molto corti rispetto alla lunghezza del testo.

Nel 1987 R. Karp ed M. Rabin hanno pubblicato un metodo (scoperto almeno dieci anni prima), detto *metodo dell'impronta randomizzato*, che mantiene l'idea primitiva dell'approccio numerico ma che è anche estremamente efficiente in quanto esso usa soltanto numeri che soddisfano l'ipotesi del modello RAM.

Questo metodo è un metodo randomizzato in cui la parte *solo se* del Teorema 4.6 continua a valere mentre la parte *se* vale soltanto *con alta probabilità*. Vediamo tale metodo nel dettaglio.

L'idea è che invece di lavorare con numeri grandi come $H(P)$ e $H(T_i)$ possiamo lavorare con tali numeri *ridotti modulo* un intero $p$ sufficientemente piccolo. Le operazioni aritmetiche si eseguono quindi su interi rappresentabili con un piccolo numero di bit e sono quindi efficienti.

Ma il principale vantaggio di questo metodo è che la probabilità di errore può essere resa piccola scegliendo casualmente $p$ in un certo insieme di valori.

**Definizione 4.7** *Dato un intero positivo $p$ indichiamo con $H_p(P)$ il resto della divisione di $H(P)$ per $p$, ossia*

$$H_p(P) = H(P) \bmod p$$

*e analogamente*

$$H_p(T_i) = H(T_i) \bmod p$$

*I numeri $H_p(P)$ ed $H_p(T_i)$ vengono detti* impronte *di $P$ e di $T_i$ rispettivamente.*

Con la riduzione di $H(P)$ ed $H(T_i)$ modulo $p$ si ottengono come impronte dei numeri compresi tra $0$ e $p - 1$ la cui dimensione rispetta il modello RAM.

Ma $H_p(P)$ ed $H_p(T_i)$ devono essere calcolati. Se per calcolarli dovessimo prima calcolare $H(P)$ ed $H(T_i)$ non avremmo risolto il problema.

Fortunatamente l'aritmetica modulare ci permette di ridurre ad ogni passo del calcolo (e non soltanto alla fine) per cui vale la seguente versione della regola di Horner:

**Lemma 4.8** *Sia $S = b_1 b_2 \dots b_m$ una stringa binaria di lunghezza $m$. Il valore di $H_p(S)$ si può calcolare nel seguente modo*

$H = b_1$
**for** $j = 2$ **to** $m$
    $H = (2H + b_j) \bmod p$

*e durante il calcolo di* $H_p(S)$ *nessun risultato intermedio è maggiore di* $2p - 1$.

Il lemma precedente ci assicura che possiamo calcolare $H_p(P)$ ed $H_p(T_i)$ per ogni $i = 1, \ldots, n - m + 1$ senza violare le ipotesi del modello RAM.

Possiamo inoltre aumentare l'efficienza osservando che per $i \geq 1$, invece di calcolare direttamente $H_p(T_i)$, possiamo calcolarlo a partire da $H_p(T_{i-1})$ con un numero di operazioni costante ed indipendente da $m$.

In questo modo il numero di operazioni per calcolare tutti gli $H_p(T_i)$ si riduce da $O(mn)$ ad $O(n)$. Di conseguenza anche il tempo calcolo per eseguire l'algoritmo si riduce da $O(mn)$ ad $O(n)$ con un notevole incremento di efficienza.

Per calcolare $H_p(T_i)$ a partire da $H_p(T_{i-1})$ osserviamo che, per ogni $i \geq 2$

$$\begin{aligned}
H(T_i) &= \sum_{j=1}^{m} 2^{m-j}\, T[i+j-1] \\
&= T[i+m-1] + \sum_{j=1}^{m-1} 2^{m-j}\, T[i+j-1] \\
&= T[i+m-1] + \sum_{j=2}^{m} 2^{m-j+1}\, T[i+j-2] \\
&= T[i+m-1] - 2^m\, T[i-1] + \sum_{j=1}^{m} 2^{m-j+1}\, T[i+j-2] \\
&= T[i+m-1] - 2^m\, T[i-1] + 2\sum_{j=1}^{m} 2^{m-j}\, T[i+j-2] \\
&= T[i+m-1] - 2^m\, T[i-1] + 2H(T_{i-1})
\end{aligned}$$

e quindi $H_p(T_i)$ si può calcolare come

$$H_p(T_i) = \{T[i+m-1] - (2^m \bmod p)\, T[i-1] + 2H_p(T_{i-1})\} \bmod p$$

Notiamo che $T[i+m-1] - (2^m \bmod p)\, T[i-1] + 2H_p(T_{i-1})$ ha un valore compreso tra $-p+1$ e $2p-1$ e e quindi si può calcolare senza violare le ipotesi del modello RAM. Inoltre $2^m \bmod p$ si può calcolare ricorsivamente, senza violare le ipotesi del modello RAM, nel modo seguente:

$$2^m \bmod p = \begin{cases} 2 & \text{se } m = 1 \\ 2\,(2^{m-1} \bmod p) & \text{se } m > 1 \end{cases}$$

e quindi ogni successiva potenza di 2 modulo $p$ ed ogni successivo valore di $H_p(T_i)$ si può calcolare in tempo costante.

L'algoritmo di Rabin e Karp è quindi il seguente:

RABIN-KARP$(P, T)$

```
1   z = 2
2   for j = 2 to m
3       z = (2z) mod p
    // z = 2^m mod p
4   x = P[1]
5   for j = 2 to m
6       x = (2x + P[j]) mod p
    // x = H_p(P)
7   y = T[1]
8   for j = 2 to m
9       y = (2y + T[j]) mod p
    // y = H_p(T_1)
10  if x == y
11      "segnala la possibile occorrenza in posizione 1"
12  for i = 2 to n − m + 1
13      y = (T[i + m − 1] − zT[i − 1] + 2y) mod p
        // y = H_p(T_i)
14      if x == y
15          "segnala la possibile occorrenza in posizione i"
```

Se $P$ occorre in $T$ in posizione $i$ chiaramente $H_p(T_i) = H_p(P)$ ma ora il viceversa può non valere. Non possiamo quindi affermare che $P$ occorre in $T$ in posizione $i$ soltanto perché abbiamo verificato che $H_p(T_i) = H_p(P)$.

**Definizione 4.9** *Se $H_p(T_i) = H_p(P)$ ma $P$ non occorre in $T$ in posizione $i$ diciamo che in posizione $i$ vi è una* falsa occorrenza *di $P$ in $T$.*

Vediamo ora come si possa scegliere un modulo $p$ che sia abbastanza piccolo da mantenere efficiente l'aritmetica ma abbastanza grande da rendere piccola la probabilità di un falsa occorrenza.

I risultati migliori si ottengono scegliendo un numero *primo* in un opportuno intervallo e sfruttando le proprietà dei numeri primi. Enunceremo le poche proprietà dei numeri primi che ci servono senza riportarne la dimostrazione.

**Definizione 4.10** *Per ogni intero positivo $n$ indichiamo con $\pi(n)$ il numero di primi minori o uguali di $n$.*

Il seguente teorema è stato dimostrato da Chebyshev nel 1859

**Teorema 4.11** *Per ogni $n \geq 11$ valgono le diseguaglianze*

$$\frac{n}{\ln(n)} \leq \pi(n) \leq 1,26\frac{n}{\ln(n)}$$

*dove $\ln(n)$ è il logaritmo naturale di $n$.*

Storicamente è stato un primo passo nella dimostrazione del famoso *teorema dei numeri primi* che afferma che

$$\lim_{n \mapsto \infty} \frac{\pi(n)}{n/\ln n} = 1$$

dimostrato da Jaques Hadamard e, indipendentemente, da Charles de la Vallé e nel 1896.

**Lemma 4.12** *Se $n \geq 29$ il prodotto di tutti i numeri primi minori o uguali ad $n$ è strettamente maggiore di $2^n$.*

Ad esempio, se $n = 29$ i numeri primi minori o uguali a 29 sono 2, 3, 5, 7, 11, 13, 17, 19, 23 e 29. Il loro prodotto è $2.156.564.410$ mentre $2^{29} = 536.870.912$.

**Corollario 4.13** *Se $n \geq 29$, qualsiasi numero $x$ minore o uguale di $2^n$ ha meno di $\pi(n)$ divisori primi distinti.*

**Dimostrazione.** Supponiamo, per assurdo, che $x$ abbia $k \geq \pi(n)$ divisori primi distinti $p_1, \ldots, p_k$. Siccome $x$ è certamente maggiore o uguale del prodotto $p_1 \ldots p_k$ dei suoi divisori primi, avremo che $2^n \geq x \geq p_1 \ldots p_k$.

Ma il prodotto $p_1 \ldots p_k$ è maggiore o uguale al prodotto dei primi $k$ numeri primi che a sua volta è maggiore o uguale al prodotto dei primi $\pi(n)$ numeri primi (in quanto $k \geq \pi(n)$).

Per il lemma precedente il prodotto dei primi $\pi(n)$ numeri primi è strettamente maggiore di $2^n$. Arriviamo quindi alla contraddizione che $2^n \geq x > 2^n$ e quindi il corollario è provato. □

Possiamo ora enunciare il teorema fondamentale per il metodo di Karp e Rabin.

**Teorema 4.14** *Siano $P$ e $T$ due stringhe di lunghezze rispettive $m$ ed $n$ tali che $mn \geq 29$ e sia $N$ un intero positivo qualsiasi maggiore o uguale di $mn$. Se $p$ è scelto casualmente tra tutti i numeri primi minori o uguali di $N$ la probabilità di una falsa occorrenza tra $P$ e $T$ è minore o uguale di $\frac{\pi(mn)}{\pi(N)}$.*

**Dimostrazione.** Sia $R$ l'insieme di tutte le posizioni in $T$ in cui *non* vi è una occorrenza del pattern $P$ ossia $H(T_i) \neq H(P)$ per ogni posizione $i \in R$.

Consideriamo il prodotto

$$\Pi = \prod_{i \in R} (|H(T_i) - H(P)|)$$

Tale prodotto deve essere minore di $2^{mn}$ in quanto $|H(T_i) - H(P)| < 2^m$ per ogni $i$ (ricordiamo che lavoriamo con stringhe binarie $S$ di lunghezza $m$ e quindi $H(S) < 2^m$).

Per il Corollario 4.13, il prodotto $\Pi$ ha al più $\pi(mn)$ divisori primi distinti.

Supponiamo che ci sia una falsa occorrenza del pattern $P$ in qualche posizione $i$ del testo $T$. Questo significa che $H(T_i) \neq H(P)$ mentre $H_p(T_i) = H_p(P)$.

Dunque $p$ è divisore della differenza $H(T_i) - H(P)$ e quindi è divisore anche del prodotto $\Pi$ e quindi $p$ è uno dei divisori primi di tale prodotto.

Siccome $p$ è stato scelto casualmente tra i primi $\pi(N)$ numeri primi, la probabilità di una falsa occorrenza è minore o uguale di $\frac{\pi(mn)}{\pi(N)}$. □

Osserviamo che il Teorema 4.14 vale per qualunque pattern $P$ e qualunque testo $T$ tali che $mn \geq 29$ e il limite $\frac{\pi(mn)}{\pi(N)}$ per la probabilità di una falsa occorrenza non dipende da $P$ e $T$ ma soltanto dalla scelta del numero primo $p$.

Quindi il Teorema non richiede alcuna ipotesi sulla distribuzione di probabilità delle stringhe $P$ e $T$ in input. Funziona per ogni $P$ e $T$ e basta!

Inoltre questo teorema non fornisce un limite superiore alla probabilità che ci sia una falsa occorrenza in una certa posizione $i$ del pattern, fornisce un limite alla probabilità che ci sia anche una sola falsa occorrenza in tutto il testo $T$.

Possiamo anche osservare che le maggiorazioni effettuate nella dimostrazione del teorema sono tutt'altro che strette e questo suggerisce che la vera probabilità di una falsa occorrenza sia in realtà molto minore del limite calcolato.

L'algoritmo delle impronte randomizzato si compone dei seguenti tre passi

1. Scegli un intero positivo $N$;
2. Scegli casualmente un numero primo $p$ minore o uguale di $N$ e calcola $H_p(P)$. (Esistono degli algoritmi randomizzati efficienti per trovare dei primi casuali come vedremo nel prossimo paragrafo).
3. Per ogni posizione $i$ di $T$ calcola $H_p(T_i)$ e controlla se è uguale ad $H_p(P)$. Se i due numeri sono uguali dichiara una possibile occorrenza oppure verifica direttamente se $P$ occorre effettivamente in $T$ nella posizione $i$.

Se escludiamo il tempo richiesto per una verifica diretta delle possibili occorrenze, il tempo richiesto dall'algoritmo è $O(m+n)$.

La scelta dell'intero $N$ è cruciale. Da una parte aumentando $N$ si riduce la probabilità di una falsa occorrenza, dall'altra si aumenta la complessità del calcolo di $H_p(P)$ e degli $H_p(T_i)$.

Sono stati proposti diversi modi di scegliere $N$ in funzione di $m$ ed $n$. Uno di questi è prendere $N = mn^2$. Con questa scelta i numeri usati nell'algoritmo richiedono al più $\log m + 2\log n + 1$ bit e quindi è soddisfatta l'ipotesi del modello RAM.

Quale è la probabilità di una falsa occorrenza in questo caso?

**Corollario 4.15** *Se $N = mn^2$ la probabilità di una falsa occorrenza è al più $\frac{2{,}53}{n}$.*

**Dimostrazione.** Per il Teorema 4.14 ed il Teorema 4.11 (dei numeri primi) la probabilità di una falsa occorrenza è limitata superiormente da

$$\frac{\pi(mn)}{\pi(mn^2)} \leq \frac{1,26\frac{mn}{\ln(mn)}}{\frac{mn^2}{\ln(mn^2)}} \leq \frac{1,26}{n}\left[\frac{\ln(m)+2\ln(n)}{\ln(m)+\ln(n)}\right] \leq \frac{1,26}{n}2 = \frac{2,53}{n} \quad \square$$

Ad esempio, prendiamo $m = 20$ ed $n = 14.000$ e quindi

$$N = mn^2 = 3.920.000.000 < 2^{32} = 4.294.967.296$$

La probabilità di una falsa occorrenza è al più $\frac{2{,}53}{14.000} = 0,00018$. Quindi usando impronte di al più 32 bit la probabilità che ci sia anche soltanto una falsa occorrenza è inferiore a 2 su 10.000.

Per molte applicazioni una probabilità di errore così piccola può essere accettata. Ad esempio in una ricerca bibliografica il fatto che due volte su 10.000 tra i libri selezionati ce ne sia qualcuno di non richiesto non è certo un problema.

Per altre applicazioni invece è indispensabile avere la sicurezza assoluta del risultato. In questo caso occorre verificare direttamente ogni possibile occorrenza e questo costa $m$ per il numero di possibili occorrenze trovate. Siccome il numero di possibili (ed anche di effettive) occorrenze può essere $\Theta(n)$ il tempo richiesto dall'algoritmo diventa $O(mn)$.

Vi è però un algoritmo che permette di verificare in tempo $O(n)$ se vi è una falsa occorrenza tra le possibili occorrenze calcolate con l'algoritmo di Karp e Rabin.

Tale algoritmo (riportato per primo da S. Mathukrishnan) ha in ingresso le posizioni delle possibili occorrenze trovate dall'algoritmo di Karp e Rabin e o risponde che non ci sono false occorrenze oppure che ce n'è almeno una (ma non quante e quali siano).

L'idea di tale algoritmo è la seguente: Consideriamo due possibili occorrenze consecutive $pos_{i-1}$ e $pos_i$ trovate dall'algoritmo di Karp e Rabin e sia $d = pos_i - pos_{i-1}$ la loro distanza. Se $d \leq m/2$ ed entrambe sono occorrenze effettive il pattern $P$ ha un bordo di lunghezza $m - d$ e quindi $d$ è un periodo del pattern e la porzione di testo $T[pos_{i-1}, pos_i + m - 1]$ ha periodo $d$ (per il Lemma 1.2). Inoltre se $P$ avesse anche un periodo proprio $p < d$ la porzione di testo $T[pos_{i-1}, pos_i + m - 1]$ dovrebbe avere periodo $p$ e dunque dovrebbe esserci una occorrenza effettiva in posizione $pos_{i-1} + p$ compresa tra le due possibili occorrenze consecutive $pos_{i-1}$ e $pos_i$; impossibile dato che l'algoritmo di Karp e Rabin trova tutte le occorrenze effettive. Quindi $d$ è il più piccolo periodo proprio del pattern $P$.

Dividiamo le posizioni delle possibili occorrenze in *corse* dove una corsa è una sequenza $pos_s, pos_{s+1}, \ldots, pos_t$ di posizioni di possibili occorrenze tale che la distanza $pos_i - pos_{i-1}$ tra due posizioni successive sia minore o uguale di $m/2$.

Se la corsa è costituita da una sola posizione l'algoritmo verifica direttamente tale posizione.

Se la corsa è più lunga l'algoritmo verifica direttamente le due prime posizioni $pos_s$ e $pos_{s+1}$. Se in una di esse vi è una falsa occorrenza risponde che ha trovato una falsa occorrenza.

Altrimenti la distanza $d = pos_{s+1} - pos_s$ è il più piccolo periodo proprio del pattern $P$ e quindi, se per qualche $i = s + 2, \ldots t$ succede che $pos_i - pos_{i-1} < p$ vi è certamente una falsa occorrenza (altrimenti $p$ non sarebbe il periodo minimo di $P$). Analogamente se succede che $pos_i - pos_{i-1} > p$ vi è pure una falsa occorrenza (altrimenti dovrebbe esserci un'altra occorrenza in una posizione intermedia).

Dunque l'algoritmo verifica se $pos_i - pos_{i-1} = p$ per ogni $i = s + 2, \ldots t$ e se trova un $pos_i - pos_{i-1} \neq p$ risponde che c'è una falsa occorrenza.

Se invece trova che tutte le distanze $pos_i - pos_{i-1}$ sono uguali a $p$ esso deve soltanto verificare che la parte $T[pos_s + m, pos_t + m - 1]$ di $T$ abbia effettivamente periodo $p$, e questo si può fare con $pos_t - pos_s - d$ confronti tra caratteri e dunque in tempo proporzionale alla lunghezza di tale sottostringa. Se tale sottostringa non ha periodo $p$ l'algoritmo risponde che c'è una falsa occorrenza, altrimenti inizia a cercare la corsa successiva.

Dopo aver verificato con successo tutte le corse l'algoritmo risponde che non ci sono false occorrenze.

L'algoritmo di Mathukrishnan completo è il seguente.

```
MATHUKRISHNAN-TEST(P, T, pos)
    // P stringa di lunghezza m. T stringa di lunghezza n ≥ m.
    // pos array delle k posizioni di possibili occorrenze trovate
    // dall'algoritmo di Rabin e Karp.
    // Se non ci sono possibili occorrenze non ci sono neanche false occorrenze.
 1  if k == 0
 2      return FALSE
    // Altrimenti controllo direttamente la prima posizione.
 3  j = 1
 4  while P[j] == T[pos[1] + j − 1]
 5      j = j + 1
 6  if j ≤ m
 7      return TRUE
 8  // Controllo le posizioni successive.
 9  i = 2
10  while i ≤ k
11      j = 1
12      while P[j] == T[pos[i] + j − 1]
13          j = j + 1
14      if j ≤ m
15          return TRUE
        // Controllo se in posizione pos[i − 1] inizia una corsa.
16      if pos[i] − pos[i − 1] ≤ m/2
            // pos[i − 1] e pos[i] sono le prime due posizioni di una corsa
            // e sono già state controllate.
17          s = i − 1, p = pos[s + 1] − pos[s]
            // Cerco la fine della corsa o una falsa occorrenza.
18          while i + 1 ≤ k and pos[i + 1] − pos[i] == p
19              i = i + 1
20          if i + 1 ≤ k and pos[i + 1] − pos[i] ≤ m/2
21              return TRUE
            // Altrimenti la corsa finisce con pos[i]. Controllo se
            // T[pos[s] + m, pos[i] + m − 1] ha periodo p
22          for j = pos[s] + m + p to pos[i] + m − 1
23              if T[j] ≠ T[j − p]
24                  return TRUE
25      i = i + 1
26  return FALSE
```

L'analisi della complessità è facile. Basta osservare che nessun carattere del testo viene confrontato più di due volte con un carattere del pattern e più di una volta con un carattere successivo del testo. Dunque il tempo richiesto è $O(n)$.

# 5 Alberi dei suffissi

Un albero dei suffissi è una struttura dati che evidenzia la struttura interna di una stringa in modo più profondo della funzione prefisso che abbiamo visto nella Sezione 2.

Gli alberi dei suffissi permettono di risolvere il problema del matching esatto in tempo lineare (al pari di altri algoritmi visti finora) ma la loro principale virtù è che essi possono essere usati per risolvere in tempo lineare molti altri problemi più complessi del pattern matching esatto.

L'applicazione classica degli alberi dei suffissi è il problema del pattern matching esatto. Dato un testo $T$ di lunghezza $n$, dopo una preelaborazione del testo che richiede tempo $O(n)$, è possibile rispondere in tempo $O(m)$ alla domanda se il pattern $P$ di lunghezza $m$ compare in $T$.

In altre parole, la preelaborazione del testo richiede tempo proporzionale alla sua lunghezza $n$ ma dopo tale preelaborazione ogni ricerca nel testo $T$ di un pattern $P$ richiede soltanto tempo proporzionale alla lunghezza $m$ del pattern cercato *indipendentemente* dalla lunghezza $n$ del testo $T$.

Descriveremo la costruzione dell'albero dei suffissi di una generica stringa $S$. Non useremo $T$ o $P$ (per testo e pattern) in quanto gli alberi dei suffissi vengono usati in un gran numero di applicazioni dove alle volte fungono da testo, alle volte da pattern, alle volte da entrambi e altre volte da nessuno dei due.

Assumeremo inoltre che l'alfabeto sia finito e noto a priori.

**Definizione 5.1** *Un albero dei suffissi $A$ per una stringa $S$ di lunghezza $n$ è un albero radicato tale che:*

1. *Ci sono esattamente $n$ foglie numerate da 1 ad $n$.*
2. *Ogni nodo interno, esclusa al più la radice, ha almeno due figli.*
3. *Ogni arco è etichettato con una sottostringa di $S$.*
4. *Due archi uscenti dallo stesso nodo non possono avere etichette che iniziano con lo stesso carattere.*
5. *La concatenazione delle etichette lungo il cammino dalla radice alla foglia numerata $i$ è esattamente il suffisso $S[i, n]$ di $S$ di lunghezza $n - i + 1$.*

Ad esempio l'albero dei suffissi per la stringa `dabdac` è quello illustrato in Figura 15. Lungo il cammino dalla radice alla foglia 1 si legge l'intera stringa `dabdac` mentre lungo il cammino dalla radice alla foglia 2 si legge il suffisso `abdac` che inizia nella posizione 2 della stringa. Il nodo $r$ è la radice ed i nodi $u$ e $w$ sono nodi interni.

La definizione precedente non garantisce l'esistenza di un albero dei suffissi per ogni stringa. Il problema è che se qualche suffisso è prefisso di un altro suffisso di $S$ il cammino relativo a tale suffisso non termina in una foglia.

Ad esempio se togliamo l'ultimo carattere `c` alla stringa `dabdac` ottenendo la stringa `dabda`, il suffisso `a` è prefisso di `abda` e quindi il cammino relativo ad `a` non termina in una foglia ma termina in un punto intermedio del cammino relativo ad `abda`.

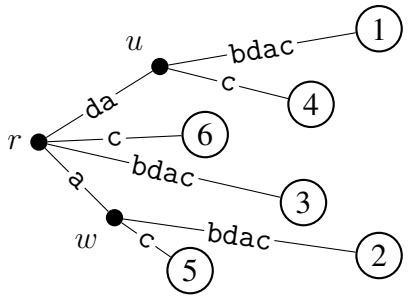



Figura 15: L'albero dei suffissi per la stringa dabdac.

Per evitare questo problema assumiamo che ogni stringa $S$ termini con un carattere diverso da tutti gli altri caratteri della stringa di modo che nessun suffisso possa essere prefisso di un altro suffisso[2].

Se ciò non è vero possiamo sempre aggiungere alla fine della stringa un carattere (sentinella) che non appartiene all'alfabeto su cui è costruita la stringa. Noi useremo il carattere \$ come sentinella e qualora sia necessario mettere in evidenza l'aggiunta della sentinella alla stringa $S$ scriveremo esplicitamente $S\$$.

**Definizione 5.2** *L'*etichetta *di un cammino è la concatenazione delle etichette degli archi del cammino.*

*L'*etichetta *di un nodo $u$ è l'etichetta del cammino dalla radice $r$ a $u$. In particolare l'etichetta della radice è la stringa nulla e l'etichetta di una foglia è il suffisso associato a tale foglia.*

*La* profondità *di un nodo è la lunghezza della sua etichetta.*

*Se l'etichetta di un arco $(u, v)$ tra due nodi $u$ e $v$ ha lunghezza $k$ maggiore di 1 l'arco $(u, v)$ è suddiviso in $k$ parti (una per ogni carattere dell'etichetta) mediante $k-1$* nodi impliciti *le cui etichette sono la concatenazione dell'etichetta di $u$ con i caratteri dell'etichetta dell'arco $(u, v)$ che precedono il nodo implicito stesso.*

Ad esempio, nella Figura 15, la stringa da è etichetta del nodo $u$ mentre la stringa dabd è etichetta di un nodo implicito interno all'arco $(u, 1)$.

## Matching esatto usando l'albero dei suffissi

Prima di entrare nel dettaglio della costruzione dell'albero dei suffissi vediamo una loro prima applicazione: la soluzione del problema della ricerca di tutte le occorrenze di un pattern $P$ di lunghezza $m$ in un testo $T$ di lunghezza $n$.

Abbiamo già visto alcuni algoritmi che risolvono il problema in modo efficiente. Gli alberi dei suffissi forniscono un altro approccio al problema basato sui seguenti passi:

1. Costruisci l'albero dei suffissi $A$ del testo $T$.

[2]Escluso ovviamente il suffisso nullo che è prefisso di ogni altro suffisso e che è rappresentato dal cammino di lunghezza zero che inizia e termina nella radice.

2. Confronta i caratteri del pattern $P$ con i caratteri dell'unico cammino in $A$ individuato da essi (ricordiamo che le etichette degli archi uscenti da un nodo interno iniziano con caratteri distinti) fino a che il pattern non sia finito oppure si arrivi ad un carattere del pattern che non compare in nessuna prosecuzione del cammino percorso fino a quel momento. In quest'ultimo caso $P$ non compare in nessuna posizione del testo $T$.

3. Se si arriva alla fine del pattern allora il pattern è uguale all'etichetta del nodo $u$ (implicito o esplicito) a cui si è arrivati e quindi $P$ è prefisso di tutti i suffissi associati alle foglie del sottoalbero radicato in $u$.

   Le posizioni di inizio di tali suffissi (che sono memorizzate nelle foglie) sono quindi tutte e sole le posizioni in cui $P$ occorre in $T$. Tali posizioni vengono quindi collezionate percorrendo il sottoalbero radicato in $u$.

Come vedremo in seguito, la costruzione dell'albero dei suffissi si può fare in tempo $O(n)$. Quindi il primo passo richiede tempo $O(n)$.

Nel secondo passo, per ogni carattere del pattern viene effettuato un solo confronto se siamo in un nodo implicito (e quindi vi è una sola possibile continuazione del cammino) e al più tanti confronti quanti sono gli archi uscenti se siamo in un nodo esplicito.

Siccome i primi caratteri delle etichette degli archi uscenti da un nodo esplicito sono tutti distinti e l'alfabeto è prefissato il numero di confronti risulta in ogni caso minore od uguale al numero di caratteri dell'alfabeto che è una costante.

Dunque è richiesto un tempo costante per ogni carattere del pattern ed il secondo passo si esegue in tempo $O(m)$.

Il terzo passo richiede tempo proporzionale al numero di nodi del sottoalbero radicato nel vertice $u$.

Se nel testo $T$ vi sono $k$ occorrenze del pattern, tale sottoalbero ha esattamente $k$ foglie. Siccome ogni nodo interno ha almeno due archi uscenti, il numero di nodi interni è minore o uguale a $k-1$ (uguale solo se ogni nodo interno ha esattamente due archi uscenti e quindi l'albero è un albero binario). Dunque il terzo passo richiede tempo $O(k)$.

Siccome $k \leq n$ il tempo totale richiesto è $O(n + m)$, uguale a quello richiesto da altri algoritmi che abbiamo visto precedentemente. Ma la distribuzione del lavoro è molto diversa.

Negli algoritmi visti finora il tempo totale era diviso in un tempo $O(m)$ per la preelaborazione del pattern $P$ ed un tempo $O(n)$ per cercare tutte le occorrenze del pattern $P$ nel testo $T$.

Invece usando l'albero dei suffissi si spende un tempo $O(n)$ per preelaborare il testo $T$ e quindi soltanto tempo $O(m + k)$ per trovare tutte le $k$ occorrenze di un qualsiasi pattern $P$ nel testo $T$.

Il vantaggio risulta chiaro nel caso in cui si debbano effettuare molte ricerche con pattern diversi in un unico grande testo.

## Un metodo ingenuo per costruire l'albero dei suffissi

Presentiamo un algoritmo ingenuo per costruire l'albero dei suffissi di una stringa $S$ allo scopo di approfondire il concetto di albero dei suffissi e sviluppare l'intuizione su di esso.

Questo algoritmo inizia la costruzione dell'albero dei suffissi con un solo arco per il suffisso $S[1, n]\$$ (l'intera stringa); quindi aggiunge uno alla volta i suffissi $S[i, n]\$$ per $i = 2, \ldots, n + 1$.

Indichiamo con $A_i$ l'albero intermedio che contiene i suffissi che iniziano nelle posizioni da 1 a $i$.

L'albero $A_1$ consiste di un unico arco etichettato con la stringa $S[1, n]\$$ che congiunge la radice ad una foglia numerata 1.

Ogni albero $A_{i+1}$ viene costruito a partire da $A_i$ nel seguente modo:

1. Partendo dalla radice di $A_i$ cerca il più lungo cammino dalla radice ad un nodo (implicito od esplicito) la cui etichetta è prefisso del suffisso $S[i+1, n]\$$ da aggiungere.

   Tale ricerca si effettua partendo dal cammino nullo (che inizia e termina nella radice e che ha la stringa nulla come etichetta) ed estendendolo, finché è possibile, aggiungendo uno alla volta i caratteri del suffisso $S[i+1, n]\$$.

   Il cammino che si ottiene in questo modo è unico in quanto il carattere successivo di un nodo implicito è unico e le etichette degli archi uscenti da un nodo esplicito iniziano con caratteri distinti.

   Inoltre l'estensione deve terminare prima della fine del suffisso $S[i+1, n]\$$ in quanto la presenza della sentinella \$ ci assicura che il suffisso $S[i+1, n]\$$ non è prefisso di nessuno dei prefissi più lunghi inseriti precedentemente nell'albero.

2. Se il nodo a cui si arriva è un nodo implicito tale nodo implicito viene sostituito con un nodo esplicito spezzando l'arco che lo contiene in due archi (e l'etichetta in due etichette).

3. A questo punto il nodo $u$ a cui siamo arrivati è un nodo esplicito con etichetta il prefisso $S[i+1, j]$ di $S[i+1, n]\$$ e tale che, nell'albero, non ci sia nessun nodo con etichetta $S[i+1, j+1]$.

   Quindi tutti gli archi uscenti da $u$ hanno etichette che iniziano con un carattere diverso dal carattere $S[j+1]$. Aggiungiamo un nuovo arco $(u, i+1)$ con etichetta $S[j+1, n]\$$ che congiunge il nodo $u$ con una nuova foglia numerata $i+1$.

   A questo punto l'albero contiene un unico cammino dalla radice alla foglia $i+1$ la cui etichetta è il suffisso $S[i+1, n]\$$ ed abbiamo quindi ottenuto l'albero successivo $A_{i+1}$.

Se assumiamo che la dimensione dell'alfabeto sia una costante, l'algoritmo ingenuo richiede tempo $O(n^2)$.

Nel caso in cui ci interessi la complessità anche in funzione del numero di simboli $|\Sigma|$ dell'alfabeto $\Sigma$ si ottiene $O(n^2\,|\Sigma|)$, che si può ridurre ad $O(n^2 \log |\Sigma|)$ usando un metodo efficiente per cercare l'arco su cui proseguire in corrispondenza dei nodi espliciti.

## L'algoritmo di Ukkonen

Il primo a scoprire un algoritmo per la costruzione dell'albero dei suffissi in tempo lineare è stato Weiner nel 1973 [7].

Per l'originalità e l'importanza del risultato tale algoritmo venne dichiarato "algoritmo dell'anno". Purtroppo negli anni successivi tale algoritmo ha ricevuto meno attenzione e uso di quanto meritasse. Probabilmente la ragione è che esso risulta difficile da capire e da implementare correttamente.

Qualche anno dopo Ukkonen [6] ha proposto un algoritmo completamente diverso. Noi esporremo quest'ultimo algoritmo per due ragioni: perché richiede meno memoria per la costruzione dell'albero dei suffissi e perchè la descrizione, dimostrazione di correttezza e analisi della complessità può essere fatta in un modo più semplice.

La semplicità è dovuta al fatto che l'algoritmo di Ukkonen può essere ottenuto partendo da un metodo semplice ed inefficiente modificandolo con alcuni *trucchi implementativi* di buon senso per renderlo più efficiente.

### Alberi dei suffissi impliciti

L'algoritmo di Ukkonen costruisce una successione di alberi dei suffissi *impliciti*, l'ultimo dei quali viene poi trasformato in un vero albero dei suffissi per la stringa $S\$$.

La definizione di albero dei suffissi implicito $I$ per una stringa $S$ è la stessa di albero dei suffissi da cui viene però rimossa la condizione che il cammino relativo ad ogni suffisso della stringa $S$ termini in una foglia ed inoltre si considera anche il suffisso nullo.

Quindi l'albero dei suffissi implicito esiste per ogni stringa $S$, anche quelle aventi dei suffissi che sono prefissi di altri suffissi e per le quali non esiste un albero dei suffissi esplicito.

Semplicemente, se un suffisso è prefisso di un altro suffisso, nell'albero dei suffissi implicito il cammino relativo a tale suffisso termina in un nodo interno (implicito od esplicito) nel cammino relativo al suffisso di cui esso è prefisso.

La relazione tra alberi dei suffissi impliciti ed alberi dei suffissi espliciti è chiarita dalla seguente

**Asserzione 1** *L'albero dei suffissi implicito di una stringa $S$ si ottiene dall'albero dei suffissi per la stringa $S\$$ togliendo il simbolo sentinella $\$$ da tutte le etichette che lo contengono, eliminando gli eventuali archi la cui etichetta si riduce alla stringa nulla e la foglia corrispondente (essi sono sicuramente archi che terminano in una foglia) ed eliminando infine gli eventuali nodi interni espliciti aventi un solo arco uscente.*

Lasciamo al lettore la semplice verifica che in questo modo si ottiene proprio l'albero dei suffissi implicito della stringa $S$.

Ci limitiamo ad illustrare la cosa con l'esempio di Figura 16 in cui è riportato l'albero dei suffissi per la stringa `dabda$` e l'albero dei suffissi implicito per la stringa `dabda` ottenuto da esso rimuovendo la sentinella $\$$.

Anche se un albero dei suffissi implicito può non avere una foglia per ogni suffisso della stringa $S$ esso contiene ugualmente tutti i suffissi di $S$ (compreso il suffisso nullo) come etichette di qualche nodo (implicito o esplicito).

L'unico problema è che se tale cammino non termina in una foglia non vi è alcuna indicazione del punto in cui tale cammino termina.

Naturalmente, se nessun suffisso (escluso quello nullo) è prefisso di un altro suffisso, l'albero dei suffissi implicito e l'albero dei suffissi esplicito coincidono.

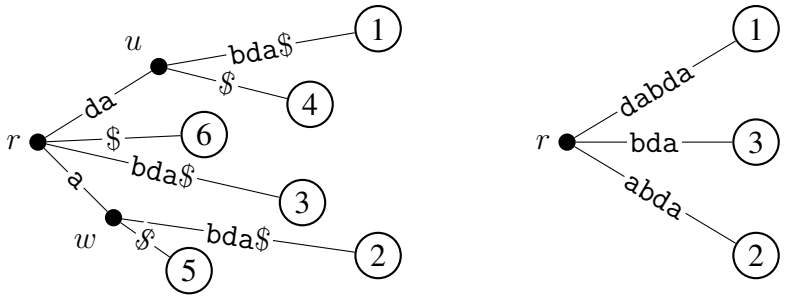



Figura 16: L'albero dei suffissi per la stringa dabda\$ e l'albero dei suffissi implicito per la stringa dabda ottenuto da esso rimuovendo la sentinella \$. Nell'albero implicito il suffisso da che inizia in posizione 4 termina in un nodo implicito interno all'arco $(r, 1)$ e il suffisso a che inizia in posizione 5 termina in un nodo implicito interno all'arco $(r, 2)$.

**Descrizione astratta dell'algoritmo di Ukkonen**

L'algoritmo di Ukkonen costruisce un albero dei suffissi implicito $I_i$ per ogni prefisso $S[1, i]$ della stringa $S\$$ di lunghezza $n{+}1$ partendo da $I_0$ e incrementando $i$ di uno finché non si arrivi a costruire $I_{n+1}$.

Siccome nessun suffisso della stringa $S\$$ è prefisso di un altro suffisso l'albero dei suffissi implicito $I_{n+1}$ coincide con l'albero dei suffissi $A$ della stringa $S\$$.

In realtà, per ottenere $A$ da $I_{n+1}$ è richiesta una trasformazione finale dovuta al fatto che per gli alberi dei suffissi impliciti si usa una rappresentazione leggermente diversa da quella usata per gli alberi dei suffissi.

La descrizione astratta dell'algoritmo di Ukkonen è quindi la seguente

UKKONEN($S$) // $S$ stringa di lunghezza $n$

1 "Aggiungi ad $S$ la sentinella ottenendo la stringa $S\$$ di lunghezza $n{+}1$."
2 "Costruisci $I_0$."
3 **for** $i = 0$ **to** $n$
4 **for** $j = 1$ **to** $i + 1$
5 "Cerca la fine del cammino relativo al suffisso $S[j, i]$ nell'albero dei suffissi implicito $I_i$ costruito al passo precedente e, se necessario, estendilo con il carattere $S[i{+}1]$ in modo tale che il suffisso $S[j, i{+}1]$ di $S[1, i{+}1]$ risulti rappresentato nell'albero."

La costruzione di $I_0$ è facile. $I_0$ ha soltanto la radice e nessun arco uscente. In esso è rappresentato l'unico suffisso del prefisso nullo $S[1, 0]$, il suffisso nullo appunto.

Vediamo quindi come eseguire l'estensione da $I_i$ ad $I_{i+1}$ ed in particolare come si effettua l'estensione di un suffisso $S[j, i]$ di $S[1, i]$ nel suffisso $S[j, i{+}1]$ di $S[1, i{+}1]$.

Abbiamo tre casi possibili:

**Caso 1:** nell'albero corrente il cammino etichettato $S[j, i]$ termina in una foglia numerata $j$. In questo caso il nuovo carattere $S[i{+}1]$ viene aggiunto all'etichetta dell'ultimo arco di tale cammino (quello che termina nella foglia).

**Caso 2:** nell'albero corrente il cammino etichettato $S[j, i]$ termina in un nodo $u$ interno (implicito o esplicito) ma nessun cammino che parte da $u$ inizia con il carattere

$S[i{+}1]$. In questo caso viene creata una nuova foglia etichettata $j$ connessa al nodo $u$ con un arco etichettato con il carattere $S[i{+}1]$. Nel caso in cui $u$ sia un nodo implicito, prima di fare ciò, il nodo $u$ viene sostituito con un nodo esplicito che suddivide in due parti l'arco a cui esso appartiene.

**Caso 3:** nell'albero corrente il cammino etichettato $S[j, i]$ termina in un nodo $u$ interno (implicito o esplicito) e almeno un cammino che parte da $u$ inizia con il carattere $S[i{+}1]$. In questo caso il suffisso $S[j, i{+}1]$ è già presente nell'albero e non occorre fare niente.

Dopo aver esteso tutti i suffissi di $S[1, i]$ siamo sicuri che nell'albero ottenuto sono rappresentati tutti i suffissi di $S[1, i{+}1]$. Infatti ogni suffisso non nullo di $S[1, i{+}1]$ è estensione di un suffisso di $S[1, i]$ ed il suffisso nullo è sempre rappresentato in ogni albero dei suffissi implicito.

Come esempio consideriamo l'albero dei suffissi implicito della stringa `adabd` riportato a sinistra nella Figura 17.

I primi quattro suffissi terminano in una foglia mentre il quinto suffisso costituito dal solo carattere `d` termina in un nodo implicito.

Se aggiungiamo un carattere `b` alla stringa i primi quattro suffissi vengono estesi usando il Caso 1; il quinto suffisso viene esteso con il Caso 2 ed il sesto suffisso, il suffisso nullo, viene esteso con il Caso 3.

A destra nella Figura 17 è riportato l'albero così ottenuto, che, come è facile verificare, è l'albero dei suffissi implicito della stringa `adabdb`.

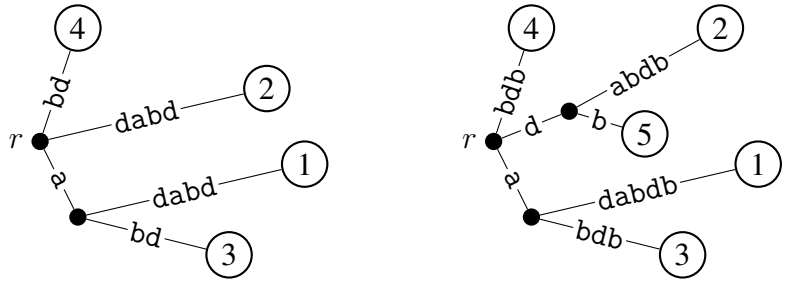



Figura 17: Estensione dell'albero dei suffissi implicito per la stringa `adabd` quando si aggiunge alla stringa il carattere `b`.

### Implementazione e miglioramento dell'efficienza

Nell'algoritmo astratto, dopo che sia stato trovato il nodo (implicito o esplicito) dove termina il suffisso $S[j, i]$ nell'albero corrente, l'estensione di $S[j, i]$ con il carattere successivo $S[i{+}1]$ richiede tempo costante in tutti e tre i Casi 1, 2 e 3.

Quindi, un punto chiave nell'implementazione dell'algoritmo di Ukkonen è trovare un metodo veloce per trovare i nodi in cui finiscono i suffissi $S[j, i]$.

In una implementazione ingenua possiamo trovare il nodo in cui termina il suffisso $S[j, i]$ in tempo $O(i - j + 1)$ partendo dalla radice dell'albero. Sommando su tutti i $j$ si ottiene un tempo $O(i^2)$ per calcolare $I_{i+1}$ a partire da $I_i$ e sommando su tutti gli $i$ si ottiene un tempo totale $O(n^3)$ per calcolare $I_{n+1}$.

Questo algoritmo sembra quindi folle poiché sappiamo che la costruzione ingenua dell'albero dei suffissi richiede tempo $O(n^2)$. Ma, come vedremo fra poco, con alcuni trucchi implementativi riusciremo a ridurre la complessità ad $O(n)$.

**Primo miglioramento: i *suffix link***

**Definizione 5.3** *Sia $\alpha$ una stringa ed $x$ un carattere. Se in un albero dei suffissi esiste un nodo interno esplicito $u$ con etichetta $x\alpha$ ed un nodo interno esplicito $s(u)$ la cui etichetta è $\alpha$, allora un* suffix link *tra $u$ ad $s(u)$ è semplicemente un puntatore da $u$ ad $s(u)$.*

Ad esempio, in Figura 15, se il nodo $u$ è il nodo di etichetta `da` ed il nodo $s(u)$ è il nodo di etichetta `a` (quello indicato con $w$ nella figura) allora tali due nodi sono collegati da un suffix link (un puntatore da $u$ a $w$) come illustrato in Figura 18.

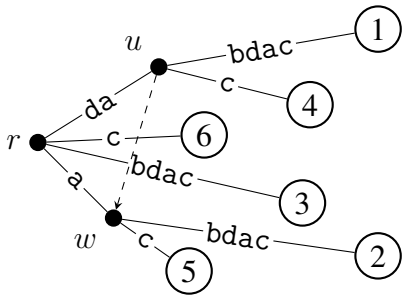



Figura 18: L'albero dei suffissi per la stringa `dabdac` con indicato il suffix link tra i nodi $u$ e $w$.

La definizione precedente, di per se, non ci assicura che da ogni nodo interno (esplicito) esca un suffix link. Con il prossimo lemma ed i suoi due corollari vedremo che ciò è sempre vero in ogni albero implicito $I_i$ costruito con l'algoritmo di Ukkonen.

**Lemma 5.4** *Se, durante la fase $i$-esima, per estendere il suffisso $S[j,i]$, viene creato un nuovo nodo interno (non foglia) con etichetta $x\alpha$ (ovviamente con $x\alpha$ prefisso di $S[j,i]$) allora o nell'albero corrente esiste già un nodo interno esplicito con etichetta $\alpha$ oppure tale nodo verrà creato immediatamente dopo con l'estensione del suffisso successivo $S[j{+}1,i]$.*

**Dimostrazione.** Un nuovo nodo interno $v$ con etichetta $S[j,i] = x\alpha$ viene creato soltanto se il suffisso $S[j,i]$ viene esteso con il Caso 2 quando nell'albero corrente il cammino etichettato $S[j,i]$ termina in un nodo interno implicito e tale cammino continua con un carattere $c$ diverso da $S[i{+}1]$.

Quindi, nell'albero corrente, il cammino di etichetta $S[j{+}1,i] = \alpha$ ha una continuazione che inizia con il carattere $c$.

Dunque il nodo $w$ con etichetta $S[j{+}1,i] = \alpha$ è un nodo interno (esplicito o implicito). Se $w$ è esplicito $s(v) = w$.

Altrimenti, se $w$ è implicito, il cammino di etichetta $S[j{+}1,i] = \alpha$ che termina nel nodo $w$ può continuare soltanto con il carattere $c$. Quindi, nel passo successivo, l'estensione del suffisso $S[j{+}1,i]$ sostituisce il nodo implicito $w$ con un nodo esplicito per cui $s(v) = w$. □

**Corollario 5.5** *Nell'algoritmo di Ukkonen, ogni nodo interno esplicito creato durante l'estensione di un suffisso ha sicuramente un suffix link da esso uscente dopo che sia stato esteso anche il suffisso successivo.*

**Dimostrazione.** La dimostrazione è per induzione sulla fase. La cosa è banalmente vera per $I_0$ in quanto esso ha un solo nodo interno esplicito, la radice che non viene creata estendendo un suffisso.

Supponiamo quindi la cosa vera fino al termine della fase $(i-1)$-esima e consideriamo la fase successiva $i$-esima.

Per il Lemma 5.4, se viene creato un nuovo nodo $v$ estendendo il suffisso $S[j, i]$, il nodo $s(v)$ viene trovato o creato durante l'estensione del suffisso successivo $S[j + 1, i]$.

Siccome l'estensione dell'ultimo suffisso (che è il suffisso nullo $S[i+1, i]$) non crea nodi interni, alla fine della fase $i$-esima tutti i nodi interni espliciti, tranne la radice, hanno il loro suffix link. □

Un altro modo per dire essenzialmente la stessa cosa è

**Corollario 5.6** *In ogni albero dei suffissi implicito $I_i$, se un nodo interno esplicito ha etichetta di cammino $x\alpha$ esiste anche un nodo interno esplicito di etichetta $\alpha$.*

Vediamo ora come si possano usare i suffix link per velocizzare l'algoritmo.

### La costruzione di $I_{i+1}$ seguendo una catena di suffix link

Ricordiamo che, nella fase $i$-esima, l'algoritmo di Ukkonen cerca i nodi in cui terminano i suffissi $S[j, i]$ di $S[1, i]$ per estendere ciascuno di essi con il carattere successivo $S[i+1]$.

Con la versione ingenua questo si fa percorrendo ogni volta un cammino che parte dalla radice dell'albero. I suffix link permettono di accorciare tali cammini ad ogni estensione.

Il suffisso $S[1, i]$ non è certamente prefisso di nessun altro suffisso (è il più lungo). Quindi il cammino relativo ad $S[1, i]$ termina in una foglia numerata 1.

Possiamo evitare di percorrere tale cammino memorizzando, nello stadio precedente, un puntatore alla foglia corrispondente al suffisso $S[1, i]$. In questo modo possiamo effettuare la sua estensione in tempo costante (applicando il Caso 1).

Supponiamo di dover estendere un suffisso successivo $S[j, i]$. Il suffisso $S[j, i]$ è uguale al suffisso precedente $S[j-1, i]$ privato del primo carattere. In altre parole se $S[j-1, i] = x\alpha$ dove $x$ è un singolo carattere e $\alpha$ è una stringa (eventualmente nulla) allora $S[j, i] = \alpha$.

Sia $v$ l'ultimo nodo esplicito del cammino relativo ad $S[j-1, i]$. Il nodo $v$ può essere o la radice oppure un nodo interno di $I_i$.

A questo punto, l'algoritmo per estendere il suffisso $S[j, i]$ deve cercare, nell'albero corrente, il nodo con etichetta $S[j, i] = \alpha$.

Se $v$ è la radice l'algoritmo si comporta come l'algoritmo ingenuo: cerca il nodo partendo dalla radice.

Se invece $v$ è un nodo interno, per il Corollario 5.6, $v$ ha un suffix link che punta ad un nodo esplicito $s(v)$. Inoltre, siccome l'etichetta del nodo $v$ è un prefisso di $x\alpha$, l'etichetta di $s(v)$ è un prefisso di $\alpha$.

Quindi per cercare il nodo con etichetta $\alpha$ possiamo partire dal nodo $s(v)$. Questa è la principale ragione per cui aggiungiamo i suffix link agli alberi dei suffissi.

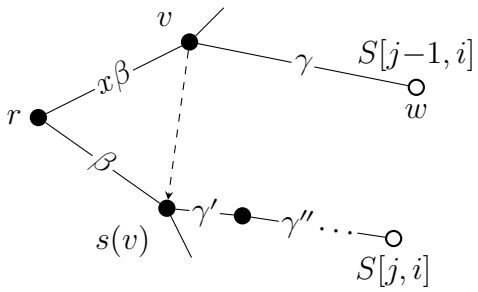



Figura 19: La ricerca del suffisso $S[j, i]$. Si risale, dal nodo con etichetta $S[j-1, i] = x\alpha = x\beta\gamma$, per al più un arco (di etichetta $\gamma$) fino al nodo $v$ di etichetta $x\beta$; ci si muove quindi in $s(v)$ di etichetta $\beta$ usando il suffix link e infine si discende cercando il cammino etichettato $\gamma = \gamma'\gamma'' \ldots$ per arrivare al nodo di etichetta $S[j, i] = \beta\gamma = \alpha$.

Più in dettaglio l'estensione del suffisso $S[j-1, i]$ avviene come segue.

Sia $\beta$ l'etichetta del nodo $s(v)$ e quindi $S[j, i] = \alpha = \beta\gamma$ (e quindi $S[j-1, i] = x\alpha = x\beta\gamma$). Per trovare la fine di $S[j, i] = \alpha$ si parte da dove finisce $S[j-1, i]$, si ritorna indietro al nodo $v$, si segue il suffix link in $v$ per arrivare al nodo $s(v)$ (che ha $\beta$ come etichetta) e quindi si discende il cammino etichettato $\gamma$ (che può essere costituito da più di un arco: $\gamma = \gamma'\gamma'' \ldots$).

La fine di questo cammino è il nodo (implicito o esplicito) con etichetta $\alpha = S[j, i]$ (vedi Figura 19). Il suffisso $S[j, i]$ viene quindi esteso in accordo con uno dei tre casi possibili.

**Algoritmo di estensione di un suffisso**

Mettendo tutto questo assieme otteniamo il seguente algoritmo per l'estensione di un suffisso:

1. Partendo dal nodo $w$ (implicito o esplicito) dove termina il suffisso $S[j, i]$ e risalendo verso la radice, cerca il primo nodo $v$ che o ha un suffix link oppure è la radice. Questo, per i Corollari 5.5 e 5.6, richiede di risalire per al più un solo arco.

   Sia $\gamma$ l'etichetta (eventualmente nulla) del cammino da $v$ al nodo $w$.

2. Se $v$ non è la radice spostati su $s(v)$ usando il suffix link e quindi discendi seguendo il cammino individuato dalla stringa $\gamma$.

   Se $v$ è la radice segui il cammino individuato da $S[j+1, i]$ partendo dalla radice (come nell'algoritmo ingenuo).

3. Estendi il suffisso $S[j+1, i]$ usando il caso appropriato.

4. Se il nodo $w$ è un nodo interno esplicito creato durante l'estensione del suffisso precedente $S[j, i]$ (usando il Caso 2), per il Lemma 5.4, la stringa $S[j+1, i]$ deve terminare in un nodo $s(w)$ esplicito (dopo l'estensione di $S[j+1, i]$).

   In questo caso metti un suffix link da $w$ ad $s(w)$.

Se assumiamo che l'algoritmo mantenga un puntatore alla foglia relativa al suffisso più lungo $S[1, i]$, la prima estensione della successiva fase $i$-esima non richiede nessuna ricerca ed inoltre essa avviene sempre usando il Caso 1.

### Che risultato abbiamo ottenuto finora?

L'uso dei suffix link è chiaramente un miglioramento rispetto al partire ogni volta dalla radice per cercare il nodo dove termina ciascun suffisso $S[j, i]$. Ma questo migliora la complessità asintotica del caso pessimo?

La risposta è negativa se l'algoritmo viene implementato nel modo descritto. Possiamo però utilizzare un semplice trucco implementativo per ridurre la complessità ad $O(n^2)$. Questo trucco è anche importante in molti altri algoritmi che usano gli alberi dei suffissi.

### Un primo trucco: lo skip/count

Nel secondo passo dell'algoritmo di estensione l'algoritmo discende dal nodo $s(v)$ lungo un cammino di etichetta $\gamma$. Se implementato direttamente questo passo richiede tempo proporzionale alla lunghezza $|\gamma|$ dell'etichetta $\gamma$.

Un semplice trucco, detto *skip/count*, permette di ridurre il tempo al numero di nodi espliciti di tale cammino e, come vedremo, questo riduce ad $O(n)$ il tempo necessario per eseguire tutta una fase e dunque ad $O(n^2)$ il tempo necessario per eseguire l'intero algoritmo di costruzione dell'albero dei suffissi.

Ecco il trucco:

**Trucco 1:** Sia $g = |\gamma|$ la lunghezza della stringa $\gamma$. Noi sappiamo che vi è sicuramente un cammino etichettato $\gamma$ che parte dal nodo $s(v)$. Se $g = 0$ il nodo cercato è proprio $s(v)$ e non dobbiamo fare niente.

Altrimenti, siccome gli archi uscenti da $s(v)$ hanno etichette che iniziano con caratteri distinti, la scelta dell'arco lungo cui scendere richiede soltanto il confronto del primo carattere di $\gamma$ con il primo carattere delle etichette degli archi uscenti da $s(v)$.[3]

Sia $b$ la lunghezza dell'etichetta $\beta$ dell'arco prescelto (che supponiamo memorizzata assieme all'etichetta).

Se $b \leq g$ l'algoritmo non deve controllare nessun altro carattere di $\beta$: $\beta$ è certamente uguale al prefisso di $\gamma$ di lunghezza $b$, ossia $\gamma = \beta\gamma'$. L'algoritmo quindi si sposta direttamente sul nodo finale dell'arco prescelto per continuare la ricerca del cammino etichettato $\gamma'$ di lunghezza $g' = g - b$.

Se $b > g$ il cammino di etichetta $\gamma$ termina nel nodo implicito $g$-esimo interno a tale arco.

[3] Questo richiede un tempo indipendente da $n$. Tale tempo dipende dalla cardinalità $|\Sigma|$ dell'alfabeto $\Sigma$, dal numero di archi $d$ uscenti da $s(v)$ e dal modo in cui sono memorizzati tali archi. Se gli archi uscenti da $s(v)$ sono memorizzati in una lista semplice il tempo è $O(d) \leq O(|\Sigma|)$. Se sono memorizzati in un albero rosso nero il tempo è $O(\log d) \leq O(\log |\Sigma|)$. L'uso di una tavola hash riduce il tempo ad $O(1)$ nel caso medio. L'uso di una tavola ad indirizzamento diretto (un array con indici i caratteri di $\Sigma$) riduce il tempo ad $O(1)$ ma richiede memoria $O(|\Sigma|)$ per ciscun nodo interno, anche quelli che hanno soltanto due archi uscenti.

Assumendo come costante la dimensione dell'alfabeto ogni spostamento da un nodo esplicito al successivo lungo il cammino etichettato $\gamma$ richiede un tempo costante (vedi Nota 3).

Questa è certamente una euristica utile. Ma quanto paga in termini di complessità asintotica. La risposta è conseguenza del prossimo Lemma 5.7.

Il *livello* $\ell(v)$ di un nodo esplicito $v$ è il numero di archi del cammino dalla radice a $v$ o, equivalentemente, il numero di nodi espliciti che precedono $v$ in tale cammino (la radice ha quindi livello 0).

**Lemma 5.7** *Quando, durante l'esecuzione dell'algoritmo di Ukkonen, viene attraversato un suffix link* $(v, s(v))$*, il livello di* $s(v)$ *è almeno pari al livello di* $v$ *meno 1, ossia* $\ell(s(v)) \geq \ell(v) - 1$.

**Dimostrazione.** Basta osservare che ogni nodo esplicito che precede $v$, esclusa la radice, ha un suffix link che punta ad un nodo che precede $s(v)$ (compresa eventualmente la radice).

Infatti se l'etichetta di cammino di $v$ è $x\beta$, prefisso del suffisso precedente $S[j, i]$, l'etichetta di cammino di $s(v)$ è $\beta$, prefisso del suffisso $S[j+1, i]$. □

Durante l'esecuzione dell'algoritmo di Ukkonen, indichiamo con $\ell$ il *livello corrente* dell'algoritmo, ossia il livello dell'ultimo nodo esplicito visitato dall'algoritmo.

**Teorema 5.8** *Usando il trucco skip/count, ogni fase dell'algoritmo di Ukkonen richiede tempo* $O(n)$.

**Dimostrazione.** Nella fase $i$-esima vengono effettuate $i+1$ estensioni. In ogni estensione l'algoritmo risale al più un solo arco per trovare il nodo con suffix link, attraversa il suffix link, discende un certo numero di archi e quindi applica uno dei tre casi per l'estensione del suffisso ed eventualmente aggiunge un suffix link.

Ciascuna di queste operazioni richiede tempo costante tranne la discesa che richiede tempo proporzionale al numero di archi percorsi.

Calcoliamo come varia il livello corrente $\ell$ durante una fase. L'estensione del primo suffisso non modifica $\ell$.

Ad ogni estensione successiva $\ell$ diminuisce di al più una unità nella risalita, diminuisce di al più una unità attraversando il suffix link ed aumenta di una unità ogni volta che si discende un arco. Quindi durante l'intera fase $\ell$ può diminuire al più $2i$ volte.

Siccome $\ell$ rimane sempre minore o uguale di $i$ (ogni arco ha una etichetta di almeno un carattere e le etichette dei cammini sono lunghe al massimo $i$) esso non può aumentare più di $3i$ volte e quindi il numero totale di archi discesi in tutta la fase è al più $3i = O(n)$. □

Siccome le fasi sono $n$ segue immediatamente che, usando i suffix link, l'algoritmo di Ukkonen si può implementare in modo da avere complessità $O(n^2)$.

A questo punto, a dispetto di tutto il lavoro fatto, non abbiamo ancora fatto nessun reale progresso rispetto all'algoritmo ingenuo che richiede anch'esso tempo $O(n^2)$.

Abbiamo però fatto un grande progresso dal punto di vista concettuale e questo ci permetterà, con soltanto ancora pochi facili passi, di ottenere un algoritmo che opera in tempo $O(n)$.

### Un semplice dettaglio implementativo: la compressione delle etichette

Vogliamo raggiungere il limite $O(n)$ per la complessità della costruzione dell'albero. Vi è però una ovvia barriera per raggiungere questo risultato: l'albero dei suffissi, come lo abbiamo rappresentato finora, può richiedere spazio $O(n^2)$.

Infatti, benché i nodi dell'albero non possano essere più di $2n-1$ la somma delle lunghezze delle etichette può essere $O(n^2)$.[4]

Siccome il tempo per costruire una struttura dati è almeno pari alla sua dimensione questo rende impossibile costruire in tempo $O(n)$ l'albero dei suffissi con la rappresentazione usata finora. Dobbiamo quindi cercare una rappresentazione per le etichette che richieda in totale spazio $O(n)$.

La soluzione è quella di rappresentare l'etichetta di un arco, non con una stringa di caratteri $\beta$ e la sua lunghezza $b$ (per operare lo skip/count) ma soltanto con una coppia di indici $(p, q)$ tale che $\beta = S[p, q]$ (la lunghezza si calcola quindi come $b = q - p + 1$). Questo è certamente possibile in quanto tutte le etichette sono sottostringhe di $S$.

Dunque ogni etichetta si può memorizzare usando uno spazio costante.[5]

### Due altri piccoli trucchi e ci siamo

Presentiamo ora altri due semplici trucchi che derivano da due osservazioni sulle interazioni tra estensioni successive nella stessa fase e in fasi successive.

La prima osservazione è la seguente:

**Osservazione 1:** Non appena, in una fase, viene applicato il Caso 3 esso viene applicato anche a tutte le estensioni successive di tale fase.

Infatti, se viene applicato il Caso 3 per l'estensione del suffisso $S[j, i]$ significa che il suffisso $S[j, i+1]$ è già presente nell'albero corrente e quindi sono presenti anche tutti i suffissi successivi da $S[j+1, i+1]$ ad $S[i+1, i+1]$.

Questa osservazione ci permette di usare il seguente

**Trucco 2:** Non appena viene applicato il Caso 3 possiamo terminare la fase.

La seconda osservazione è la seguente:

**Osservazione 2:** Se ad un certo punto dell'algoritmo di Ukkonen viene creata una foglia etichettata $j$ per inserire il suffisso $S[j, i+1]$ tale foglia rimane una foglia in tutti i successivi alberi costruiti dall'algoritmo.

Infatti l'algoritmo non estende mai un cammino oltre la foglia che lo termina.

Quindi, non appena abbiamo costruito una foglia etichettata $j$, in tutte le fasi successive l'estensione del suffisso $S[j, i]$ viene effettuata usando il Caso 1.

[4] I nodi sono le $n$ foglie ed al più $n-1$ nodi interni. Nel caso di una stringa con tutti i caratteri distinti l'albero dei suffissi è costituito soltanto dalla radice e dalle $n$ foglie e la somma delle lunghezze delle etichette è $\sum_{i=1}^{n} i = O(n^2)$.

[5] Facciamo la solita assunzione del modello di calcolo RAM: che i numeri rappresentabili con meno di $\log n$ bit si possono memorizzare, leggere, scrivere, sommare, sottrarre, moltiplicare, dividere e confrontare usando tempo e spazio costante.

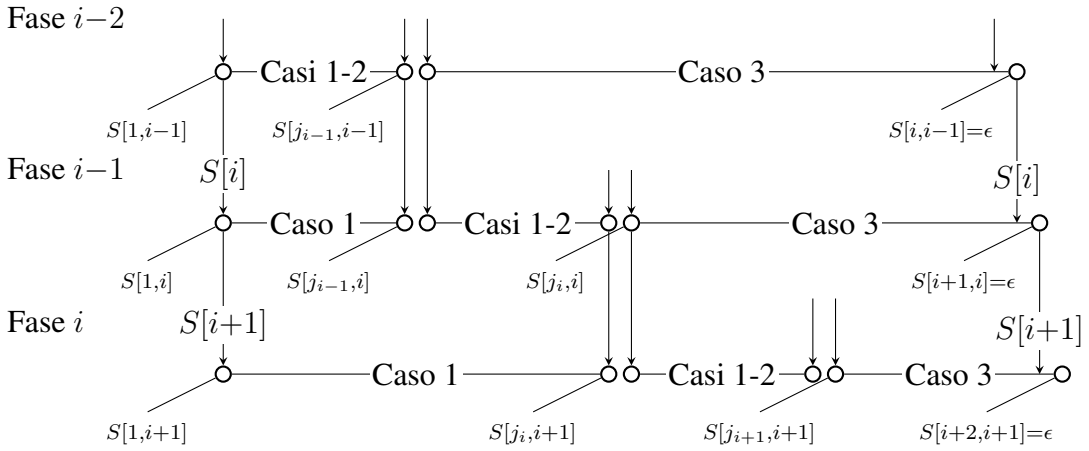



Figura 20: Rappresentazione dell'esecuzione di due fasi successive dell'algoritmo di Ukonnen: la fase $i-1$ che estende i suffissi di $S[1,i-1]$ ottenuti nella fase $i-2$ con il carattere $S[i]$ e la fase $i$ che estende ulteriormente i suffissi con il carattere $S[i+1]$. Alla fine di ogni fase sono elencati i nodi finali (impliciti o espliciti) dei suffissi nell'ordine in cui essi sono stati estesi e tra essi è evidenziato il nodo relativo all'ultimo suffisso al quale è stato applicato il Caso 1 o 2 (rispettivamente i nodi relativi ai suffissi $S[j_{i-1},i-1]$, $S[j_i,i]$ e $S[j_{i+1},i+1]$). Alla fine dell'elenco dei nodi di ogni fase viene aggiunto il suffisso vuoto $\epsilon$ (che non si ottiene per estensione di un suffisso precedente).

La foglia 1 viene costruita nella fase 1. Quindi ogni fase inizia con una estensione che usa il Caso 1, continua con estensioni che usano i Casi 1 e 2 e termina non appena viene usato il Caso 3.

Sia $j_{i-1}$ l'ultima estensione che usa i Casi 1 o 2 nella fase $(i-1)$-esima. Siccome ogni applicazione del Caso 2 crea una nuova foglia, alle prime $j_{i-1}$ estensioni della fase successiva si applica il Caso 1 (per l'Osservazione 2).

La Figura 20 illustra la situazione relativamente a due fasi successive.

Dunque $j_i \geq j_{i-1}$ e questo ci suggerisce di cercare un trucco che faccia evitare, nella fase successiva, le prime $j_{i-1}$ estensioni che usano il Caso 1.

A questo scopo osserviamo che il Caso 1 si limita ad aggiungere all'etichetta dell'arco che termina nella foglia di etichetta $S[j,i]$ l'ultimo carattere $S[i+1]$ del suffisso $S[j,i+1]$ che stiamo inserendo.

Quando viene creata una foglia con il Caso 2 nella fase $i$-esima (o direttamente nella fase 1) all'arco che termina nella foglia viene assegnato il singolo carattere $S[i+1]$ come etichetta (rappresentata dalla coppia di indici $(i+1,i+1)$). Da quel momento in poi l'etichetta di tale arco verrà allungata con un singolo carattere successivo ad ogni fase (con l'applicazione del Caso 1) fino a diventare $S[i+1,n]$ nell'ultima fase.

Con la nostra rappresentazione delle etichette questo significa incrementare il secondo indice ad ogni fase fino ad arrivare al valore $n$.

Il trucco è quindi:

**Trucco 3:** Se la lunghezza $n$ della stringa $S$ è nota a priori, si assegna direttamente la

coppia $(i{+}1, n)$ come etichetta dell'arco entrante in ogni foglia creata nella fase $i$-esima.

Se la lunghezza $n$ della stringa $S$ non è nota a priori (ricordiamo che l'algoritmo di Ukkonen è on-line) si assegna la coppia $(i{+}1, e)$ dove $e$ è un valore speciale diverso da ogni indice (ad esempio $e = 0$) e che, in ogni fase successiva, rappresenta la fine corrente di tale etichetta (che è $i$ durante la fase $i$-esima ed $n$ alla fine).

In questo modo le prime $j_{i-1}$ estensioni nella fase $i$-esima non richiedono alcuna operazione e possono essere saltate iniziando quindi direttamente con l'estensione $(j_{i-1}{+}1)$-esima.

Quindi la fase $i$-esima inizia con l'estensione $j = j_{i-1} + 1$ e termina con la prima estensione che usa il Caso 3 oppure quando tutti i suffissi sono stati estesi e quindi $j = i{+}2$.

A questo punto basta porre $j_i = j - 1$ per avere il valore corretto di $j_i$ per la fase successiva.

**Teorema 5.9** *Usando i suffix link ed i Trucchi 1, 2 e 3, l'algoritmo di Ukkonen richiede tempo $O(n)$ per costruire tutti gli alberi dei suffissi impliciti da $I_0$ ad $I_n$.*

**Dimostrazione.** Se $S[j^*, i{-}1]$ è l'ultimo suffisso esteso nella fase $(i{-}1)$-esima, la fase $i$-esima inizia con l'estensione del suffisso $S[j^*, i]$ (o con l'ultimo suffisso $S[i{+}1, i]$ se nella fase $(i{-}1)$-esima non si è applicato mai il Caso 3).

Dunque soltanto il suffisso $j^*$-esimo viene esteso sia nella fase $(i{-}1)$-esima che nella fase $i$-esima.

Siccome gli indici $j$ dei suffissi sono tutti minori o uguali ad $n$ e le fasi sono $n$, in totale vengono eseguite al più $2n$ estensioni durante tutta l'esecuzione dell'algoritmo di Ukkonen.

Ricordiamo che ogni estensione richiede un tempo costante più un tempo proporzionale al numero di archi discesi nella ricerca del cammino etichettato $\gamma$ e che, nella dimostrazione del Teorema 5.8, abbiamo visto che il numero totale di archi discesi durante una intera fase è al più $3i = O(n)$.

Per dimostrarlo abbiamo usato il fatto che passando dall'estensione di un suffisso all'estensione del suffisso successivo non si risalgono mai più di due archi.

Siccome la fase $i$-esima inizia con il suffisso $j^*$-esimo in cui è avvenuta l'ultima estensione della fase precedente, il livello corrente non cambia passando da una fase alla successiva (anche nel caso in cui $j_{i-1} = i$, il livello corrente quando si è esteso l'ultimo suffisso $S[i, i{-}1] = \epsilon$ nella fase $(i{-}1)$-esima era 0, uguale al livello corrente quando si estende il primo suffisso $S[i{+}1, i] = \epsilon$ nella fase $i$-esima).

Dunque, il fatto che il livello corrente non diminuisca mai più di due volte passando dall'estensione di un suffisso all'estensione del suffisso successivo vale non soltanto all'interno di una singola fase ma anche durante tutta l'esecuzione dell'algoritmo.

Siccome il livello rimane sempre compreso tra 0 ed $n$ il numero di archi discesi durante tutta l'esecuzione dell'algoritmo deve essere minore o uguale di $3n$.

Siccome durante tutta l'esecuzione dell'algoritmo di Ukkonen vengono eseguite al più $2n$ estensioni ciascuna delle quali richiede un tempo costante più un tempo proporzionale agli archi discesi e il numero totale di archi discesi durante tutto l'algoritmo di Ukkonen è minore o uguale a $3n$ possiamo concludere che il tempo totale richiesto è $O(n)$. □

Se la lunghezza $n$ della stringa $S$ era nota a priori e quindi non abbiamo usato l'indice speciale $e$, l'ultimo albero implicito $I_n$ è esattamente l'albero dei suffissi esplicito che volevamo costruire, altrimenti per ottenerlo basta percorrere, in tempo $O(n)$, l'albero $I_n$ sostituendo ogni occorrenza dell'indice speciale $e$ nelle foglie con il valore $n$ (che dopo aver elaborato tutti i caratteri della stringa $S$ è ovviamente noto).

Dunque vale il seguente

**Teorema 5.10** *L'algoritmo di Ukkonen costruisce, on-line, l'albero dei suffissi di una stringa $S$ di lunghezza $n$ in tempo $O(n)$.*

### Esercizi

**Esercizio 10** Costruire una famiglia infinita di stringhe su di un alfabeto finito prefissato, tale che la somma delle lunghezze delle etichette degli archi dei loro alberi dei suffissi cresca più rapidamente di $O(n)$ (dove $n$ è la lunghezza della stringa).

**Esercizio 11** Studiare la relazione tra l'albero dei suffissi di una stringa $S$ e quello della stringa rovesciata $S^R$ (quella che si ottiene "leggendo" $S$ alla rovescia). Suggerimento: cominciare pensando alla relazione tra i suffix link dei due alberi.

# 6 Alcune applicazioni degli alberi dei suffissi

## 6.1 Problema dello string matching esatto

Vi sono tre importanti varianti nell'uso degli alberi dei suffissi per il pattern matching esatto in dipendenza di quale delle due stringhe $P$ e/o $T$ è nota per prima.

Abbiamo già visto l'uso nel caso in cui è noto sia il pattern che il testo. In tal caso l'uso di un albero dei suffissi permette di risolvere il problema in tempo $O(m + n)$, lo stesso degli algoritmi di Knuth, Morris e Pratt e di Boyer e Moore (dove $m$ è la lunghezza del pattern ed $n$ è la lunghezza del testo).

Ci sono però situazioni in cui il testo $T$ è noto e rimane invariato per un lungo tempo durante il quale viene richiesto di cercare le occorrenze in $T$ di un gran numero di pattern $P$. In questo caso, dopo aver preprocessato il testo, vogliamo poter trovare le occorrenze di un pattern $P$ nel minor tempo possibile.

Se usiamo l'albero dei suffissi di $T$ (che si costruisce nella fase di preelaborazione di $T$ in tempo $O(n)$) ogni successiva ricerca di un pattern $P$ richiede soltanto un tempo $O(m+k)$ totalmente indipendente dalla lunghezza $n$ del testo ($k$ è il numero di occorrenze di $P$ in $T$).

La situazione opposta, in cui è noto il pattern $P$ viene gestita in modo ottimale dagli algoritmi di Knuth, Morris e Pratt e di Boyer e Moore che spendono tempo $O(m)$ per preprocessare il pattern e quindi tempo $O(n)$ per ricercarlo in un testo qualsiasi $T$ di lunghezza $n$.

Possiamo ottenere lo stesso risultato anche usando gli alberi dei suffissi (anche se con un valore maggiore della costante nascosta dalla notazione asintotica). In questo caso, gli alberi dei suffissi per risolvere il problema del matching esatto non sono convenienti.

Il loro uso diventa però utile se dobbiamo risolvere un problema un poco più generale: Trovare per ogni posizione $i$ del testo la lunghezza della più lunga sottostringa del pattern che occorre nel testo in posizione $i$.

## 6.2 Problema del set matching esatto

Abbiamo visto come il problema del set matching esatto (trovare tutte le occorrenze di un insieme di pattern $\mathcal{P}$ nel testo $T$) si possa risolvere in modo efficiente usando l'algoritmo di Aho e Corasick.

L'algoritmo di Aho e Corasick richiede tempo $O(m+n+k)$ (dove $m$ è la somma delle lunghezze dei pattern in $\mathcal{P}$, $n$ è la lunghezza del testo $T$ e $k$ è il numero di occorrenze).

Lo stesso risultato si può ottenere costruendo dapprima l'albero dei suffissi di $T$ in tempo $O(n)$ e quindi cercando uno alla volta in $T$ ciascun pattern $P$ della famiglia $\mathcal{P}$ in tempo $O(|P| + k_P)$ (dove $k_P$ è il numero di occorrenze di $P$), per un tempo totale $O(m + k)$.

## 6.3 Problema dello string matching su di un insieme di testi

In questo problema è noto e prefissato un certo insieme di testi $\mathcal{T} = \{T_1, \ldots, T_k\}$ (una base dati testuale). Dopo di che, per un grande numero di pattern $P$ dobbiamo cercare tutti i testi nella base dati che contengono il pattern $P$ come sottostringa.

Questo è il problema inverso del set matching dove si cercano le occorrenze dell'insieme $\mathcal{P}$ di pattern nella stringa testo $T$.

Gli alberi dei suffissi permettono di risolvere il problema in un modo molto semplice. Si costruisce una albero dei suffissi generalizzato $\mathcal{A}$ che contiene i suffissi di tutti i testi della base dati.

L'albero $\mathcal{A}$ si può costruire in tempo $O(n)$ (dove $n$ è la somma delle lunghezze dei testi) usando una semplice modifica dell'algoritmo di Ukkonen che invece di partire da un albero con la sola radice per inserire tutti i suffissi di una stringa parte dall'albero che contiene tutti i suffissi delle stringhe precedentemente inserite.

L'etichetta di una foglia di $\mathcal{A}$ può essere suffisso di uno o più testi della base dati. Ad ogni foglia restano quindi associati uno o più suffissi dei testi.

Possiamo anche costruire con l'algoritmo di Ukkonen l'albero dei suffissi della stringa $S = T_1\$_1T_2\$_2 \ldots T_k\$_k$ dove $\$_1,\ \$_2, \ldots, \$_k$ sono sentinelle distinte. Alla fine occorre sostituire l'etichetta $i$ di ogni foglia con una coppia $i', j$ in cui $j$ è l'indice del testo in cui cade la posizione $i$ di $S$ e $i'$ è la posizione di inizio del suffisso nella stringa $T_j\$_j$. Occorre inoltre sostituire l'indice fittizio $e$ dell'arco che termina nella foglia di etichetta $i', j$ con la lunghezza della stringa $T_j\$_j$.

In questo caso inoltre, avendo scelto sentinelle tutte distinte, ogni foglia è etichettata con il suffisso di uno solo dei pattern $T_j\$_j$.

La ricerca delle stringhe che contengono il pattern $P$ procede quindi come usuale partendo dalla radice.

Se non non si trova nessun cammino etichettato $P$ il pattern $P$ non compare in nessun testo della base dati.

Altrimenti il pattern $P$ compare in tutte i testi della base dati associati alle foglie del sottoalbero radicato nel nodo di etichetta $P$.

Il tempo per cercare $P$ di lunghezza $m$ è $O(m + k)$, indipendente dalla dimensione della base dati.

## 6.4 Problema della più lunga sottostringa comune

Il problema richiede di cercare la più lunga sottostringa comune tra due stringhe $S_1$ ed $S_2$ di lunghezze $n_1$ ed $n_2$.[6]

Il problema si può risolvere in modo efficiente e concettualmente semplice costruendo l'albero dei suffissi generalizzato per le due stringhe $S_1$ ed $S_2$ (tempo $O(n_1 + n_2)$) e marcando i nodi con 1 e/o 2 se il sottoalbero radicato in tale nodo ha almeno una foglia associata alla stringa $S_1$ e/o $S_2$ (ricerca in profondità: tempo $O(n_1 + n_2)$). Infine le sottostringhe comuni sono le etichette di tutti i nodi marcati sia 1 che 2 e una semplice percorrenza dell'albero permette di determinare la più lunga (ancora tempo $O(n_1 + n_2)$).

### Esercizi

**Esercizio 12** Supponiamo che il pattern $P$ contenga alcuni caratteri jolly, ossia dei caratteri speciali che stanno per un qualsiasi altro carattere. Ad esempio

$$P = \texttt{ab}\#\texttt{cad}\#\#\texttt{ababb}\#$$

dove $\#$ è il carattere jolly.

Il pattern $P$ occorre nella posizione $i$ del testo $T$ se le due stringhe $P$ e $T[i, i + m - 1]$ coincidono nelle posizioni dei caratteri non jolly.

Trovare un algoritmo che trova tutte le occorrenze del pattern $P$ con jolly nel testo $T$ in tempo $O(n + m)$.

# Riferimenti bibliografici


[1] A. Aho, and M. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. ACM*, 18:333–340, 1975.

[2] R. Baeza-Yates, and G. Gonnet. A new approach to text searching. *Comm. ACM*, 35:74–82, 1992.

[3] R.S. Boyer, and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[4] R. Karp, and M. Rabin. Efficient randomized pattern matching algorithms. *IBM J. Res. Development*, 31:249–260, 1987.

[5] D.E.Knuth, J.H. Morris, and V.B Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.


[6] Questo è un problema completamente diverso dalla ricerca della più lunga sottosequenza comune, che si risolve con la programmazione dinamica in tempo $O(n_1 n_2)$.

[6] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.

[7] P. Weiner. Linear pattern matching algorithms. *Proc. of the 14th IEEE Symp. on Switching and Automata Theory*, pp. 1–11, 1973.