Teoria della Computazione

Tommaso Ferrario (@TommasoFerrario18)

Telemaco Terzi (@Tezze2001)

October 2023

Indice

Capitolo 1

Introduzione

La teoria della complessità computazionale nasce col fine di classificare il problemi in base alla difficoltà delle soluzioni mediante macchine di calcolo. Tale difficoltà viene stimata rispetto a **spazio** e **tempo**. La teoria della complessità computazionale si riferisce a varie classi di complessità che classificano i problemi nel seguente modo:

- **Facili** o **trattabili**: ovvero i problemi che sono risolvibili in modo *efficiente*.
- **Difficili**: ovvero i problemi che si possono risolvere con algoritmi ma *non* in modo efficiente e non ho una dimostrazione che mi assicuri che non siano risolvibili in modo efficiente.
  - **Difficilissimi**: ovvero quei problemi che se riuscissi a risolvere con un algoritmo efficiente renderebbero tutti i problemi difficili facili. (*Riduzioni polinomiali*)
- **Dimostrabilmente difficili** o **Dimostrabilmente intrattabili**: ovvero i problemi che possono essere risolti ma so che *non* esiste un algoritmo efficiente in quanto è stato dimostrato che non può esistere.
- **Impossibili** o **indecidibili**: non so risolverli sempre neanche in modo non efficiente, ossia esiste almeno un input per cui l'algoritmo non funziona, ma esiste almeno un caso in cui funzioni.

Riuscire ad associare un problema rispetto a una delle classi appena presentate, apre la possibilità di risolverlo utilizzando i metodi sviluppati per la risoluzione degli altri problemi appratente a quella classe.

Definizione 1 (**Problema Computazionale**)**.** *Un **problema computazionale** è definito da un **input**, ossia come sono fatti i dati su cui opero, un **output** ovvero come sono fatti i risultati e una **relazione** tra input e output.*

$$\Pi \subseteq I \times O \tag{1.1}$$

Definizione 2 (**Algritmo**)**.** *Un **algoritmo** è una procedura composta da un numero finito di operazioni concrete e generali tale che se applicato a una istanza calcola l'output del problema.*

Gli algoritmi possono essere distinti in:

- ***Algoritmo efficiente**: che mi da la soluzione in **tempo polinomiale** rispetto alla dimensione dell'input. Ho un caso peggiore limitato superiormente:*

  $$\mathcal{O}(p(n)) \tag{1.2}$$

  Ho una crescita di tempo accettabile all'aumentare dell'input.

- ***Algoritmo non efficiente**: che mi da la soluzione ma in tempo superiore a quello polinomiale. Ho un caso peggiore limitato superiormente da un esponenziale:*

  $$\mathcal{O}(2^n) \tag{1.3}$$

  Ho una crescita di tempo assolutamente non accettabile all'aumentare dell'input.

Esempio 1 (**Problema dell'Arco minimo**)**.** *Dato in input un grafo pesato sugli archi $G = (V, E)$. Come output, voglio ottenere l'arco con peso minimo.*

Una possibile soluzione consiste nel scorrere tutti gli archi e selezionare quello di peso minimo. Dato che per implementare questa soluzione basta iterare su tutti gli archi si ha un tempo paria a $\mathcal{O}(n)$ (in realtà $\theta(n)$), quindi in tempo lineare sul numero di archi (è quindi in tempo polinomiale).

Esempio 2 (**Problema di raggiungibilità**). *Dato in input un grafo non pesato $G = (V, E)$ e due vertici, uno sorgente e uno destinazione, tali che v_s, $v_d \in V$. In output, voglio sapere se posso arrivare a v_d partendo dal vertice v_s.*

La soluzione più semplice consiste nell'analizzare tutti i cammini che partono da v_s e posso dare la risposta. Tale soluzione richiede un tempo pari a $\mathcal{O}(2^{|E|})$. Il tempo quindi cresce in modo esponenziale. Una soluzione migliore è quella di usare un algoritmo di visita (BFS o DFS) che richiede tempo $\mathcal{O}(|V|+|E|)$, ovvero un tempo polinomiale. Quindi per quanto all'inizio si pensi che sia un problema difficile si scopre che è un problema facile.

Esempio 3 (**Problema TSP (*Travelling Salesman Problem*)**). *Dato in input un grafo pesato sugli archi e completo $G = (V, E)$. Voglio ottenere il cammino minimo che tocca tutti i vertici del grafo una e una sola volta.*

Sarebbe facile determinare un ciclo ma non quello di peso minimo e per farlo devo trovare tutti i cicli e trovare quello di peso minimo. Ho quindi un algoritmo che è $\mathcal{O}(2^n)$ (nella realtà è circa $\mathcal{O}(n!)$ che è comunque esponenziale). In questo caso non si riesce a pensare ad una soluzione che non sia esponenziale nel tempo (anche se per alcuni input sia di facile risoluzione, basti pensare ad avere tutti gli archi di peso 1, ma mi basta avere un input problematico). Non potendo però dimostrare che sia irrisolvibile si dice che è un problema difficilissimo.

Esempio 4 (**Problema Dimostrabilmente Intrattabile**). *Dato in input un insieme V di n elementi, si vogliono enumerare tutti i sottoinsiemi di V.*

In questo caso l'algoritmo migliore è esponenziale dato che l'output desiderato è esponenziale.

$$|V| = n \text{ allora } |P(V)| = 2^n \tag{1.4}$$

È importante osservare che la complessità computazionale deve essere valutata tenendo in considerazione l'input e considerando sempre il caso peggiore. Oltre a ciò anche la rappresentazione che si utilizza influenza il valore della complessità computazionale.

Capitolo 2

Macchina di Turing

2.1 Macchina di Turing

La **macchina di Turing** consiste di un controllo finito che può trovarsi in un stato, scelto in un insieme finito. Tale macchina è composta da un **nastro**, potenzialmente infinito, diviso in **celle**, ognuna delle quali può contenere un simbolo scelto in un insieme finito chiamato **alfabeto**.

L'input è una stringa di lunghezza finita formata da simboli scelti dall'*alfabeto* di input (Σ), e viene inizialmente posto sul nastro. In tutte le altre celle, che si estendono sia a destra che a sinistra senza limiti, è presente il simbolo *blank* $\sqcup$, il quale specifica che in quella posizione non è presente un simbolo dell'alfabeto. C'è però un eccezione, infatti, all'inizio della sequenza di input è presente il simbolo $\triangleright$, il quale indica la posizione dell'input sul nastro.

Definizione 3 (**Macchina di Turing**). *La **macchina di Turing** è definita come una quadrupla:*

$$M = (Q, \Sigma, q_0, \delta) \tag{2.1}$$

dove:

- Q *è un insieme finito di stati.*
- Σ *è un alfabeto finito al quale sono aggiunti due caratteri di controllo:*
  1. $\triangleright$*: indica il punto di partenza della sequenza di input.*
  2. $\sqcup$ *(blank): il quale è presente in tutte le celle del nastro, escluse quelle contenenti l'input, nell'istante di partenza.*
- q_0 *rappresenta lo stato iniziale.*
- δ *è la funzione di transizione definita come:*

$$\delta : Q \times \Sigma \longrightarrow Q \times \Sigma \times \{\rightarrow, \leftarrow, -\} \tag{2.2}$$

  Tale funzione esprime il comportamento passo per passo della macchina di Turing. Essa prende in input uno stato e un simbolo, e restituisce come output una tripla composta da un nuovo stato, il simbolo scritto nella posizione indicata dalla testina e lo spostamento della testina.

  In base allo stato in cui mi trovo e al valore presente sotto la testina si applica la funzione di transazione.

Osservazione 1. *Dato che l'alfabeto dei simboli è finito, la funzione di transizione è definibile, ovvero è un algoritmo.*

La macchina di Turing si arresta quando non ho più transazioni valide oppure quando entra in uno stato accettante.

Per esprimere la computazione di una macchina di Turing usiamo una **configurazione**, ovvero sulla base della definizione della macchina di Turing e dello stato attuale devo definire tutti i passi.

La **configurazione** descrive in ogni istante lo stato della macchina, viene rappresentata da una quadrupla definita come:

$$(q,\ \text{simbolo sotto la testina},\ \text{stringa a sinistra della testina},\ \text{stringa a destra della testina}) \tag{2.3}$$

Descriviamo le **mosse** della macchina di Turing $M = (Q, \Sigma, q_0, \delta)$ mediante la notazione $\vdash$.

Definizione 4 (**Computazione**). *Una sequenza di configurazioni in cui si trova la macchina prende il nome di* ***computazione****.*

Teorema 1 (**Tesi di Church-Turing**). *Se un problema è umanamente calcolabile, allora esisterà una macchina di Turing in grado di risolverlo.*

È una tesi che non ha dimostrazione formale ma è stata dimostrata empiricamente nel corso degli anni. Portando quindi a dire che il calcolo è ciò che può essere eseguito con un Macchina di Turing. Quindi ciò che è computabile è computabile da una macchina di Turing o da un suo equivalente.

Esempio 5 (**Successore**). *Si scriva la macchina di Turing che calcoli il successore di un numero binario, che sarà l'input (e si da per scontato che sia correttamente formattato avendo solo 0 o 1 come simboli). Si trascuri il riporto (nel senso che non aggiungo ulteriori bit).*

Definisco quindi la macchina di Turing come:

- $Q = \{ini, incr, uno, zero, H\}$
- $\Sigma = \{1, 0, \triangleright, \sqcup\}$
- *La funzione di transizione come:*

$$\begin{aligned}(ini, 0/1) &\to (ini, 0/1, \to)\\ (ini, \sqcup) &\to (incr, \sqcup, \leftarrow)\\ (incr, 0) &\to (H, 1, -)\\ (incr, 1) &\to (incr, 0, \leftarrow)\\ (incr, \triangleright) &\to (uno, \triangleright, \to)\\ (uno, 0) &\to (zero, 1, \to)\\ (zero, 0) &\to (zero, 0, \to)\\ (zero, \sqcup) &\to (H, 0, -)\end{aligned} \tag{2.4}$$

- *ini come lo stato iniziale.*

Ogni operazione sulla macchina di Turing ha lo stesso tempo e quindi posso usare il numero di passi per calcolare il tempo di risoluzione. Il tempo di calcolo di una macchina di Turing è definito come il numero di configurazioni che la macchina M attraversa quando riceve in input un valore x. Questo valore si indica come:

$$t_M(x) \tag{2.5}$$

Più in generale possiamo esprimere i tempi di calcolo come:

$$T_M(n) = \max_{x \in \Sigma^*} \{t_M(x) \mid |x| = n\} \tag{2.6}$$

ovvero sto cercando il tempo del caso peggiore tra tutti gli input della stessa dimensione.

2.1.1 Macchina di Turing multi-nastro

Per semplificare le computazioni svolte con la macchina di Turing, è possibile definire una macchina che utilizza più nastri. Tale macchina viene chiamata **macchina multi-nastro** nella quale sono presenti k nastri, ognuno dei quali può essere utilizzato per operazioni di lettura e scrittura.

Questa macchina ha una testina per ogni nastro, la quale è associata alla cella su cui voglio eseguire le operazioni.

Esempio 6 (**Decidere se una stringa è nella forma** $a^n b^n c^n$). *Vogliamo decidere se la stringa in input è del tipo $a^n b^n c^n$ con k nastri. Per risolvere questo problema è possibile utilizzare una macchina di multi-nastro. Ad esempio, utilizzando 3 tre nastri posso inserire la stringa di input su tutti e tre e posizionare le testine all'inizio delle sequenze di a, quella sul secondo nastro all'inizio della sequenza di b e quella sul terzo all'inizio della sequenza di c.*

Una macchina di Turing a k nastri **non** è più potente di una macchina di Turing a singolo nastro. Esiste sempre una traduzione verso una macchina di Turing a singolo nastro.

Teorema 2 (**Equivalenza tra macchina mono nastro e macchina multi nastro**). *Sia M una macchina di Turing con k nastri, allora esiste una macchina M' a nastro singolo equivalente, ovvero che mi permette di ottenere lo stesso risultato.*

Dimostrazione 1. *Una semplice metodologia per passare da una macchina di Turing multi nastro a una con singolo nastro, consiste nel concatenare gli input nel singolo nastro e separarli con l'ausilio di uno specifico carattere. Oltre a ciò, posso introdurre una variante dei simboli dell'alfabeto in modo da poter indicare le posizioni delle k testine sul singolo nastro. Infine, è necessario aumentare il numero degli stati, passando da $|Q|$ a $|Q| \times |\Sigma|^k$ per rappresentare le varie configurazioni.*

Teorema 3 (**Equivalenza tra macchina multi nastro e macchina mono nastro tempi di calcolo**). *Se alla macchina M è associata una funzione di complessità $T_M(n)$, allora alla macchina M' corrisponde una funzione $T_{M'}(n) = \mathcal{O}((T_M(n))^2)$.*

Dimostrazione 2. *Se la lunghezza dei k nastri è $\leq t_M(x)$ allora la lunghezza della macchina mono-nastro sarà $\leq k \cdot t_M(x)$. Per eseguire la scansione del nastro sarà richiesto un tempo pari a $\mathcal{O}(k \cdot t_M(x))$, dato che $t_M(x)$ è il tempo di esecuzione del programma, allora il tempo di calcolo della macchina mono-nastro sarà pari a:*

$$\mathcal{O}(t_M(x)) \cdot t_M(x) = \mathcal{O}((t_M(x))^2) \tag{2.7}$$

Data una macchina di Turing M con alfabeto Σ, tale che $|\Sigma| > 4$, esiste sempre una macchina di Turing M' equivalente tale che $|\Sigma| = 4$, ovvero che risolve lo stesso problema con l'utilizzo di un **alfabeto binario**.

$$\forall x \in \Sigma \; M(x) = M'(f(x)) \tag{2.8}$$

Per convertire i simboli di un alfabeto Σ in binario è possibile utilizzare la seguente conversione:

$$\sigma \in \Sigma \text{ posso rappresentarlo con } \log_2 |\Sigma| = k \tag{2.9}$$

Il tempo di esecuzione richiesto per risolvere lo stesso problema utilizzando un alfabeto binario è lo stesso della macchina con un alfabeto composto da più simboli.

$$t_M(x) \;\rightarrow\; t_{M'}(f(x)) = \mathcal{O}(k \cdot t_M(x)) = t_{M'}(f(x)) = \mathcal{O}(t_M(x)) \tag{2.10}$$

2.1.2 Macchina di Turing universale

Esiste una macchina di Turing che mi permette di eseguire macchine di Turing. Tale macchina è chiamata **macchina universale**, indicata con il simbolo U. La macchina universale U è tale che $\forall \alpha, x \in \Sigma^*$, allora $U(\alpha, x) = M_\alpha(x)$ dove M_α è la macchina di Turing rappresentata da α.

Posso rappresentare U come una macchina con tre nastri, dove il primo contiene la descrizione della macchina M_α, il secondo contiene l'input della macchina M_α. Il terzo contiene le informazioni relative allo stato corrente di M_α.

Il tempo di calcolo di tale macchina dipende dalla dimensione del programma:

$$t_U(\alpha, x) = \mathcal{O}(|\alpha| \cdot t_{M_\alpha}(x)) \tag{2.11}$$

2.2 Linguaggi

Le macchine di Turing calcolano funzioni del tipo:

$$f : \Sigma^* \to \Sigma^* \tag{2.12}$$

Oltre a questo, una macchina di Turing può decidere un linguaggio fornendo una risposta binaria ($\{0, 1\}$), ovvero data una stringa in input è sempre in grado di dire se appartiene o meno al linguaggio.

Definizione 5 (**Linguaggio ricorsivo**). *Un linguaggio è **decidibile** o **ricorsivo**, se esiste almeno una macchina di Turing che **decide** il linguaggio, fermandosi sempre in 1 o 0. Solitamente è un linguaggio finito, ma potrebbe anche essere infinito.*

$$M(x) = \begin{cases} 1 & \text{se } x \in L \\ 0 & \text{altrimenti} \end{cases} \tag{2.13}$$

Definizione 6 (**Linguaggio ricorsivamente enumerabile**). *Un linguaggio è **semi-decidibile** o **ricorsivamente enumerabile**, se esiste almeno una macchina di Turing che **accetta** il linguaggio:*

$$M(x) = \begin{cases} 1 & \text{se } x \in L \\ 0 \vee \ \perp & \text{se } x \notin L \end{cases} \tag{2.14}$$

dove $\perp$ rappresenta il fatto che una macchina di Turing non termina.

Quindi, un linguaggio ricorsivamente enumerabile è un linguaggio per cui data una stringa in input la macchina di Turing si ferma, altrimenti la macchina di Turing potrebbe o fermarsi o andare avanti all'infinito nella computazione.

Teorema 4 (**Complemento di un lingiaggio ricorsivo è ricorsivo**). *Sia L un linguaggio ricorsivo allora $\overline{L}$ è ricorsivo.*

Dimostrazione 3. *Dato che L è un linguaggio ricorsivo, allora esiste sempre una macchina che mi decide il linguaggio, ovvero che mi restituisce 1 se $x \in L$ e 0 altrimenti. $\overline{L}$ è il lingiaggio che contiene tutte le stringhe che non appartengono a L. Quindi, posso definire una macchina che mi decide $\overline{L}$ partendo da quella che mi decide L e invertendo gli output.*

Teorema 5 (**Linguaggio ricorsivo è anche ricorsivamente enumerabile**). *Preso un linguaggio L costruito alfabeto Σ si ha che se $L \subseteq \Sigma^*$ se L è ricorsivo è anche ricorsivamente enumerabile.*

Dimostrazione 4. *Se L è ricorsivo esiste una macchina di Turing che riconosce se, data una stringa x, tale che $x \in L$, rispondendo 1 e risponde 0 se $x \notin L$.*

Costruisco ora una macchina di Turing M' che se il linguaggio non è ricorsivo allora vado in loop ($\perp$) Quindi quando la macchina sta per andare in 0 modifico δ per ottenere il loop, ottenendo una macchina che va in loop se $x \notin L$, mentre restituisce 1 se $x \in L$ e quindi L è ricorsivamente enumerabile.

Teorema 6. *L è ricorsivo se e solo se L è ricorsivamente enumerabile e il complementare di L, $\overline{L}$ è ricorsivamente enumerabile.*

Dimostrazione 5. *Rispetto alla dimostrazione precedente, che garantisce che se L è ricorsivo allora è ricorsivamente enumerabile, devo definire, per il complementare, una macchina di Turing che va in loop se $x \in L$. Presa quindi una macchina di Turing M che decide L, definisco M' che accetta L, che quindi è ricorsivamente enumerabile. Definisco ora M'' che restituisce Y se $x \in \overline{L}$ ovvero $x \notin L$ ed entra in loop se $x \notin \overline{L}$, ovvero $x \in L$.*

Quindi M decide L eseguendo alternativamente M' e M'' prestando attenzione alla gestione degli output.

Vogliamo ora dimostrare che esistono dei problemi di decisione che non possono essere risolti da una macchina di Turing. Per realizzare questa dimostrazione partiamo dal fatto che $|A| < |\mathcal{P}(A)|$, ovvero che la cardinalità dell'insieme A è sempre minore della cardinalità dell'insieme delle parti di A ($\mathcal{P}(A)$).

Con le informazioni ottenute in precedenza abbiamo visto che possiamo rappresentare una macchina di Turing partendo dall'alfabeto binario definito come $B = \{0, 1\}$. Da questo possiamo affermare che il numero di macchine di Turing che possiamo realizzare sarà al più il numero di stringhe realizzabili con l'utilizzo di B, ovvero $|B^*|$:

$$\#M = |B^*| \tag{2.15}$$

Inoltre, utilizzando l'alfabeto B, possiamo definire il linguaggio di decisione come:

$$L_D = \{x \in B^* \mid f_D(x) = 1\} \subseteq B^* \tag{2.16}$$

quindi, possiamo affermare che il numero di problemi di decisione che si possono avere è uguale a $|P(B^*)|$. Partendo dal concetto matematico prima presentato, possiamo affermare che:

$$\#M \leq |B^*| < |P(B^*)| \tag{2.17}$$

ovvero che esistono problemi di decisione che *non posso* risolvere con una macchina di Turing.

2.2.1 Halting Problem

Un esempio di problema di decisione che non può essere risolto da una macchina di Turing è l'**halting problem**. Questo problema consiste nel determinare se esiste un algoritmo che dati in input una macchina di Turing M e una stringa x, mi dica se la macchina M eseguita sull'input x termina.

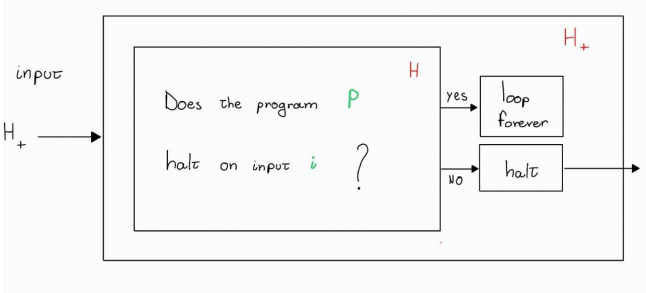

Figura 2.1: Rappresentazione grafica della dimostrazione dell'Halting problem

Definizione 7. *Posso definire formalmente l'**halting problem** con l'utilizzo del linguaggio L_H, definito come:*

$$L_H = \{M \cdot \# \cdot x \mid M(x) \neq \perp\} \tag{2.18}$$

ovvero come l'insieme di stringhe per cui M termina. Si utilizza il carattere $\#$ per separare la descrizione della macchina e l'input.

Teorema 7 (**Halting problem non è decidibile**)**.** *Il linguaggio L_H non è ricorsivo e quindi l'halting problem non è decidibile.*

Dimostrazione 6 (*Per assurdo*)**.** *Assumiamo, per assurdo, che L_H sia ricorsivo. Quindi esiste una macchina di Turing M_H che prende in input $(M \cdot \# \cdot x)$ e mi fornisce in output:*

$$M_H(M \cdot \# \cdot x) = \begin{cases} Y & se\ M(x)\ \neq \perp \\ N & se\ M(x)\ = \perp \end{cases} \tag{2.19}$$

Se esiste tale macchina, allora posso costruire una macchina di Turing C, costruita partendo da M_H che prende in input M e mi restituisce:

$$C(M) = \begin{cases} Y & se\ M_H(M, M) = N \\ N & se\ M_H(M, M) = Y \end{cases} \tag{2.20}$$

A questo punto se fornisco alla macchina C se stessa come input ottenendo due possibili situazioni:

- *$C(C) = Y$ allora $M_H(C, C) = N$ ma quindi mi aspetto che $C(C) = \perp$ il che mi porta ad un assurdo.*
- *$C(C) = \perp$ allora $M_H(C, C) = Y$ ma quindi mi aspetto che $C(C) \neq \perp$ il che mi porta ad un assurdo.*

Posso quindi affermare che non esiste una macchina M_H che decide L_H.

Teorema 8 (**Halting problem è ricorsivamente enumerabile**)**.** *Il linguaggio L_H è ricorsivamente enumerabile, ovvero il problema è parzialmente decidibile.*

Dimostrazione 7. *Esiste una macchina di Turing M_A che accetta il linguaggio L_H, ovvero tale che $M_A(M, x) = Y$ se e solo se $M(x) \neq \perp$.*

In precedenza abbiamo definito la macchina di Turing universale, la quale $U(M, x) = M(x)$, allora:

$$M_A(M, x) = \begin{cases} Y & se\ e\ solo\ se\ U(M, x) = Y\ \vee\ N \\ \perp & se\ U(M, x) = \perp \end{cases} \tag{2.21}$$

Esiste quindi una macchina che accetta il linguaggio L_H.

Teorema 9. *$\overline{L_H}$ non è ricorsivamente enumerabile.*

Dimostrazione 8 (*Per assurdo*)**.** *Assumiamo per assurdo che $\overline{L_H}$ sia ricorsivamente enumerabile. Dai teoremi precedenti sappiamo che L_H è ricorsivamente enumerabile, allora per quanto definito in precedenza si ha che L_H è ricorsivo, il che è assurdo in quanto è stato dimostrato in precedenza che L_H non è ricorsivo.*

2.3 Problemi decidibili

Vogliamo ora studiare i problemi decidibili classificandoli sulla base dei tempi di esecuizione, considerando la dimensione dell'input e il caso peggiore. Per fare questo utilizzeremo il tempo di esecuzione di una macchina di Turing. Questo valore viene calcolato tramite il numero di passi che la macchina compie.

Definizione 8 (**Tempo di calcolo**). *Definiamo $t_M(x)$ come il tempo di calcolo di una macchina di Turing M su input x. Non è un caso peggiore ma dipendente dal singolo input specifico. Il tempo di calcolo è il numero di passi che esegue M su input x per dare una risposta.*

Non si usa comunque il numero di passi nella realtà ma si usa la notazione $\mathcal{O}$-grande, studiando il caso peggiore, ovvero il numero massimo di passi.

Definizione 9 (**Complessità temporale**). *Definiamo $T_M(n)$ come la* ***funzione di complessità temporale*** *come:*

$$T_M(n) = \max\{t_M(x) \mid |x| = n\} \tag{2.22}$$

Definizione 10 (**Classe P**). *Definisco la classe P in base alle macchine di Turing. La classe P è definita come:*

$$P = \{L \mid L \text{ è deciso da una macchina di Turing in tempo polinomiale } \mathcal{O}(p(n))\} \tag{2.23}$$

Definizione 11 (**Classe DTIME**). *Definiamo la classe $DTIME(f(n))$ come la classe dei linguaggi decisi da una macchina di Turing entro un tempo $f(n)$:*

$$DTIME(f(n)) = \{L \subseteq \Sigma^* \mid \exists M \text{ decide } L \text{ in tempo } \mathcal{O}(f(n))\} \tag{2.24}$$

Quindi $DTIME(n)$ rappresenta l'insieme dei problemi di decisione che possono essere risolti con un algoritmo che lavora in tempo $\mathcal{O}(n)$. Quindi:

$$P = \bigcup_{c \in \mathbb{N}} DTIME(n^c) \tag{2.25}$$

Infatti P è l'unione di tutte le classi DTIME con funzioni polinomiali.

$$DTIME(n) \subseteq DTIME(n^2) \subseteq DTIME(2^{n^c}) \tag{2.26}$$

Possiamo anche definire la classe **EXPTIME** come la classe di linguaggi decidibili da una macchina di Turing in tempo esponenziale:

$$EXPTIME = \bigcup_{c \in \mathbb{N}} DTIME(2^{n^c}) \tag{2.27}$$

Inoltre, si ha che vale la seguente relazione tra le classi P e $EXPTIME$:

$$P \subseteq EXPTIME \tag{2.28}$$

Teorema 10. *Se un problema è nella classe P allora è risolvibile in un tempo efficiente.*

Devo anche definire la complessità spaziale oltre a quella temporale.

Definizione 12 (**Spazio**). *Definisco lo* ***spazio*** *di calcolo $s_M(x)$ come il numero di celle del nastro usate dalla macchina di Turing M con input x durante la computazione.*

Il calcolo non è semplice come per il tempo, avendo anche decrementi. Quindi più che "celle usate" studiamo le "celle visitate".

Definizione 13 (**Complessità spaziale**). *Definisco $S_M(n)$ come la* ***funzione di complessità spaziale****:*

$$S_M(n) = \max\{s_M(x) \mid |x| = n\} \tag{2.29}$$

Esiste una relazione tra la complessità spaziale e quella temporale per una macchina di Turing. Se una computazione dura n passi di tempo, allora posso dire che al più ho usato n celle di spazio, questo perché è possibile che in alcune configurazioni la testina non si muove, ma nel caso peggiore si sposta sempre. Si ha quindi:

$$S_M(n) \leq T_M(n) + n \tag{2.30}$$

con $+ n$ perché sul nastro abbiamo comunque l'input di lunghezza n.

Teorema 11. *Se il tempo è limitato allora lo spazio è limitato ma non vale l'opposto.*

Teorema 12. *Se ho una macchina di Turing M che lavora in spazio finito e tempo infinito, esiste una macchina di Turing M' che fa la stessa cosa di M in tempo limitato. Quindi se lo spazio è limitato allora il tempo è limitato.*

Dimostrazione 9. *Infatti la macchina M' può trovarsi in un numero finito di stati K e avendo spazio limitato ho un numero limitato $S_M(n)$ di celle in cui si trova la testina. Ho anche un numero finito di simboli in alfabeto Σ e quindi:*

$$S_{M'}(n) \leq |k| \cdot |S_M(n)| \cdot |\Sigma|^{|S_M(n)|} \tag{2.31}$$

avendo che prima o poi ritorno a stati già visti quindi la macchina se supera la quantità appena definita capisce di essere in loop. Quindi $|k| \cdot |S_M(n)| \cdot |\Sigma|^{|S_M(n)|}$ è anche un limite temporale per la seconda macchina.

Quindi data una certa macchina che lavora in un certo spazio $S(n)$ posso costruire una macchina equivalente che da la stessa risposta in tempo limitato $T(n)$. Si ha che se ho un problema che si risolve in spazio polinomiale, per la formula appena scritta avrò tempo esponenziale. Invece, al contrario, tempo polinomiale comporta spazio polinomiale.

2.4 Macchine di Turing non deterministiche

Una **macchina di Turing non deterministica** si distingue da quella deterministica nella funzione di transizione δ. Nella macchina non deterministica la funzione di transizione associa a ogni stato q e a ogni simbolo di nastro X un *insieme* di triple:

$$\delta(q, X) = \{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\} \tag{2.32}$$

A ogni passo una macchina di Turing non deterministica sceglie una delle triple come mossa, ovviamente lo stato, il simbolo di nastro e la direzione appartengono alla stessa tripla. Nelle macchine non deterministiche la computazione non è una sequenza di configurazioni ma un albero di computazione. Da ogni stato posso passare a uno tra più stati, a seconda della scelta, formando così un albero. Il singolo passo di computazione non è univocamente definito. Ogni singolo ramo comunque è equivalente al passo di computazione della macchina di Turing deterministica.

Definizione 14 (**Linguaggio accettato**). *Un linguaggio L è **accettato** da una macchina di Turing non deterministica N se per tutte le stringhe che fanno parte del linguaggio esiste almeno una computazione che termina nello stato Y, ovvero esiste una computazione per cui:*

$$\forall x \in L \Rightarrow N(x) = Y \tag{2.33}$$

Nel caso in cui nessuna delle computazioni termina nello stato Y, allora l'input non è accettato.

Definizione 15 (**Linguaggio deciso**). *Un linguaggio L è **deciso** da una macchina di Turing non deterministica N se, qualora la stringa x appartenga il linguaggio, esiste almeno una computazione tale per cui:*

$$x \in L \rightarrow N(x) = Y \tag{2.34}$$

altrimenti, se x non appartiene al linguaggio, per tutte le computazioni, si ha che:

$$x \notin L \rightarrow N(x) = N \tag{2.35}$$

Non devo quindi avere loop in questo caso, tutte devono dare N.

Teorema 13 (**Equivalenza tra macchine di Turing deterministica e non deterministica**). *Se M_N è una macchina di Turing non deterministica, esiste una macchina di Turing deterministica M_D tale che $L(M_N) = L(M_D)$.*

Dimostrazione 10. *La macchina di Turing deterministica M_D simula la macchina di Turing non deterministica M_N procede con una visita in ampiezza delle configurazioni in modo da evitare di incappare in loop.*

La macchina di Turing non deterministica N decide il linguaggio L in tempo $t_N(n)$, ovvero il tempo di calcolo è dato dall'altezza del ramo più lungo. Come per le macchine deterministiche, definiamo $T_N(n)$ come:

$$T_N(n) = \max\{t_N(x) \mid |x| = n\} \tag{2.36}$$

Data una macchina di Turing non deterministica N che decide L in tempo $T_N(n)$ esiste una macchina di Turing deterministica M che decide L in tempo $2^{\mathcal{O}(T_N(n))}$. Questa complessità deriva dal fatto che se ogni nodo dell'albero ha al massimo b figli, allora l'albero di computazione ha al massimo $b^{T_N(n)}$ foglie. Il numero interno di nodi è al massimo $b^{T_N(n)} - 1$ e quindi il numero totale di nodi è $< 2 \cdot b^{T_N(n)}$. Inoltre, un cammino dalla radice alla foglia è $\mathcal{O}(T_N(n))$. Il tempo di esecuzione della macchina M è $\mathcal{O}(T(n) \cdot b^{T(n)})$ che può essere visto come $2^{\mathcal{O}(T_N(n))}$.

Definizione 16 (**Classe NTIME**). *Definiamo una funzione di tempo per una macchina di Turing non deterministica come:*

$$NTIME = \{L \mid L \text{ è deciso da una macchina di Turing non deterministica in tempo } \mathcal{O}(f(n))\} \tag{2.37}$$

Quest'ultima definizione ci permette di definire la classe di **problemi NP** come l'insieme dei linguaggi L che sono decisi in tempo polinomiale da una macchina di Turing non deterministica:

$$NP = \bigcup_{c>0} NTIME(n^c) \tag{2.38}$$

Osservazione 2. *Sia P l'insieme dei **problemi decidibili in tempo polinomiale** da una macchina di Turing deterministica, e NP l'insieme dei **problemi decidibili in tempo polinomiale** da una macchina di Turing non deterministica. Sappiamo che è vero che $P \subset NP$ ma non è ancora dimostrato che $NP \subset P$.*

Possiamo quindi affermare che:

$$P \subseteq NP \subseteq EXPTIME \tag{2.39}$$

NP rappresenta una classe di linguaggi che sono verificabili in tempo polinomiale da una macchina di Turing non deterministica. Verificare un linguaggio significa che per ogni parola $w \in L$ si ha una stringa c detta **certificato** che possiamo usare per verificare che effettivamente $w \in L$.

Definizione 17. *Un **verificatore** di un linguaggio L è una macchina di Turing deterministica V tale per cui:*

$$L = \{w : V \text{ accetta } \langle w, c\rangle \text{ per qualunque stringa } c\} \tag{2.40}$$

Questa macchina accetta le stringhe appartenenti al linguaggio eseguendo le istruzioni specificate dal certificato c.

Se V richiede tempo polinomiale rispetto w a per accettare/rifiutare, allora V è un verificatore in tempo polinomiale per w. Se esiste un verificatore in tempo polinomiale per L, allora L è verificabile in tempo polinomiale.

Dato che V deve eseguire in tempo polinomiale rispetto a w, ne segue che c, il certificato, deve essere di lunghezza polinomiale rispetto a w, altrimenti non avremmo il tempo di leggere tutto c.

Supponiamo di avere una macchina di Turing non deterministica M che lavora in tempo polinomiale, costruiamo un verificatore V che lavora in tempo polinomiale per lo stesso linguaggio deciso da M. Se su input w la macchina M accetta, significa che esiste una computazione accettante $C_1, \ldots, C_k$ di lunghezza polinomiale rispetto a w. Per ogni passaggio da C_i a C_{i+1} è stata applicata una transizione definita dalla funzione di transizione di M.

Usiamo come certificato c la sequenza di transizioni applicate lungo tutta la computazione, le quali sono in numero polinomiale. Queste transizioni ci identificano una specifica computazione della macchina di Turing non deterministica M. Il verificatore deve solo simulare quella computazione, senza fare scelte non deterministiche, dato che le transizioni da fare sono tutte in c e verificare che la computazione accetti.

Abbiamo mostrato che per ogni linguaggio accettato da una macchina di Turing non deterministica che lavora in tempo polinomiale possiamo costruire un verificatore polinomiale per lo stesso linguaggio. Dobbiamo mostrare anche l'inclusione in senso inverso. Sfruttiamo il fatto che possiamo usare una macchina di Turing non deterministica per scrivere un certificato sul nastro e, se esiste un certificato che ci permette di accettare, questo sarà presente in una computazione.

Sia V un verificatore in tempo polinomiale per L. Assumiamo V esegua in tempo n^k. Costruiamo una macchina di Turing non deterministica che su input w fa due cose:

- Genera in modo non deterministico un certificato c di lunghezza al più n^k.
- Simula V su input $\langle w, c\rangle$ e accetta se V accetta, mentre rifiuta se V rifiuta.

Abbiamo mostrato che per ogni linguaggio per il quale esiste un verificatore polinomiale possiamo costruire una macchina di Turing non deterministica che decide lo stesso linguaggio in tempo polinomiale. Quindi le due definizioni di NP sono equivalenti.

Definiamo ora altre classi di problemi:

- Linguaggi di cui posso decidere la **non appartenenza** in tempo polinomiale:
$$coP = \{L \mid \overline{L} \in P\} \tag{2.41}$$
- Linguaggi che **non sono decidibili** in tempo polinomiale: $\overline{P}$
- Linguaggi di cui posso decidere la **non appartenenza** in tempo polinomiale da una macchina di Turing non deterministica:
$$coNP = \{L \mid \overline{L} \in NP\} \tag{2.42}$$

Nota 1.

$$\overline{P} \neq coP \tag{2.43}$$
$$P \subseteq NP \ \wedge \ P \subseteq coNP \text{ quindi } P \subseteq NP \cap coNP \tag{2.44}$$

Teorema 14.

$$P = coP \tag{2.45}$$

Dimostrazione 11. *Se un linguaggio $L \in P$, allora esiste una macchina di Turing deterministica che decide L in tempo polinomiale:*

$$\forall x \in \Sigma^* = \begin{cases} x \in L \Rightarrow M(x) = Y \\ x \notin L \Rightarrow M(x) = N \end{cases} \tag{2.46}$$

Posso inoltre creare una macchina M' che decide $\overline{L}$ in tempo polinomiale:

$$\forall x \in \Sigma^* = \begin{cases} x \in \overline{L} \Rightarrow M'(x) = Y \\ x \notin \overline{L} \Rightarrow M'(x) = N \end{cases} \tag{2.47}$$

per ottenere questa macchina è sufficiente modificare gli stati di accettazione e rifiuto della macchina M.

La dimostrazione precedente non può essere utilizzata per dimostrare che $NP = coNP$ dato che, nel caso di macchine non deterministiche è sufficiente che esista un singolo ramo di computazione che termina in uno stato di accettazione. Invertendo l'output di questa macchina si ottiene una macchina che non decide $\bar{L}$ dato che sarebbero necessaire tutte computazioni che terminano in uno stato accettante.

2.5 Riduzioni polinomiali

Le **riduzioni polinomiali** tra problemi sono delle procedure che per ogni istanza del problema A la trasformano in un'istanza per un problema diverso B. Quindi un'istanza di A, I_A, ha due risposte, Y o N, ma posso passare tramite una determinata funzione:

$$f : I_A \to I_B \tag{2.48}$$

avendo poi I_B tale che B avrà risposte, uguali a quelle di A, Y o N. In altre parole, si cerca una funzione f che mi permette di convertire le istanze del mio problema di partenza in istanze di un problema che so risolvere.

Definizione 18 (**Riduzione polinomiale**). *La **riducibilità polinomiale** tra problemi è definita come:*

$$f : \Sigma^* \to \Sigma^* \tag{2.49}$$

la quale deve essere calcolabile in tempo polinomiale da una macchina di Turing deterministica, tale che:

$$\forall x \in \Sigma^*, \ x \in L_A \text{ se e solo se } f(x) \in L_B \tag{2.50}$$

Indichiamo l'operazione di riduzione polinomiale come:

$$L_A \leq_P L_B \tag{2.51}$$

Osservazione 3. *Si utilizza il simbolo minore uguale, per indicare una relazione d'ordine nella complessità dei problemi.*

Teorema 15. *Dato un linguaggio L_A riducibile polinomialmente a L_B ($L_A \leq_P L_B$), si ha che se $L_B \in P$ allora sicuramente anche $L_A \in P$.*

Dimostrazione 12. *Infatti e esiste un algoritmo polinomiale per L_B allora, avendo una trasformazione polinomiale ho che $f(x) + L_B$ è ancora polinomiale.*

Teorema 16. *La riduzione polinomiale gode della proprietà di transitività, ovvero:*

$$L_A \leq_P L_B \wedge L_B \leq_P L_C \text{ allora } L_A \leq_P L_C \tag{2.52}$$

Posso ottenere questa riduzione applicando la composizione di funzioni.

Definizione 19 (**NP-hard**)**.** *Un linguaggio L è **NP_hard** se vale che:*

$$\forall L' \in NP \text{ si ha } L' \leq_P L \tag{2.53}$$

Definizione 20 (**NP-completo**)**.** *Un linguaggio L si dice **NP_completo** se valgono i seguenti punti:*

- $L \in NP$
- $L \in NP\_hard$

Teorema 17. *Se $L_A \leq_P L_B$ e $L_A \in NP\_hard$ allora so che anche $L_B \in NP\_hard$.*

Dimostrazione 13. *Se $L_A \leq_P L_B$ per la definizione di NP_hard so che esistono n linguaggi che sono riducibili polinomialmente a L_A. Utilizzando la proprietà transitiva posso affermare che L_B è NP_hard.*

Teorema 18 (**Teorema Cook-Levin**)**.** *Il problema di soddisfacibilità SAT è $NP\_completo$. Questo problema prende in input una formula booleana ϕ in forma normale congiunta (CNF), ovvero che ha una congiunzione ($\wedge$) come legame tra le clausole. Una clausola è un $\vee$ di letterali, ovvero di variabili booleane x_i o $\overline{x_i}$. In output ho se la forma sia soddisfacibile o meno.*

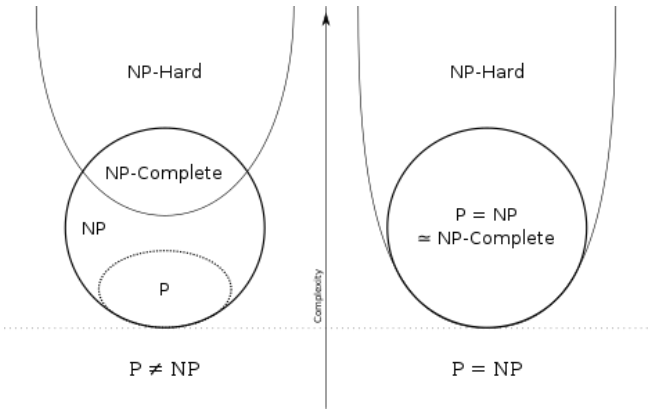

Figura 2.2: Classificazione dei problemi.

Capitolo 3

Strutture dati succinte

Le **strutture dati succinte** sono una classe di strutture dati che forniscono un compromesso tra l'efficienza dell'accesso ai dati e la quantità di spazio utilizzato per memorizzare i dati. Queste strutture cercano di minimizzare l'uso dello spazio di memoria, mantenendo nel contempo un accesso efficiente ai dati.

Le strutture dati succinte sono un tipo di struttura dati che utilizza una quantità di spazio "vicina" al limite inferiore teorico dell'informazione, ma che, a differenza di altre rappresentazioni compresse, consente ancora operazioni di interrogazione efficienti.

Supponiamo che $\mathcal{Z}$ sia il numero ottimale teorico di bit necessari per memorizzare alcuni dati. Una rappresentazione di questi dati viene chiamata:

- **Implicita** se richiede $\mathcal{Z} + \mathcal{O}(1)$ bit di spazio. (es. $\mathcal{Z} + 14$ bit)
- **Succinta** se richiede $\mathcal{Z} + o(\mathcal{Z})$ bit di spazio. (es. $\mathcal{Z} + \log \mathcal{Z}$ oppure $\mathcal{Z} + \sqrt{\mathcal{Z}}$ bit)
- **Compatta** se richiede $\mathcal{O}(\mathcal{Z})$ bit di spazio. (es. $5 \cdot \mathcal{Z}$ bit)

Nota 2. *$o(\mathcal{Z})$ si riferisce al concetto matematico di o-piccolo, ovvero:*

$$\lim_{x \to x_0} \frac{f(x)}{g(x)} = 0 \Rightarrow f(x) = o_{x_0}(g(x)) \ \textit{con}\ x_0 = +\infty \tag{3.1}$$

3.1 Bitvector

Un modo di rappresentare le strutture dati succinte è attraverso l'utilizzo di **bitvector**.

Definizione 21 (**Bitvector**)**.** *Si definisce un **bitvector** B come un array di lunghezza n, popolato da elementi binari ($\{0, 1\}$). Formalmente si ha:*

$$B[i] \in \{0, 1\}, \ \forall i \ \textit{tale che}\ 1 \leq i \leq n \tag{3.2}$$

In alternativa all'alfabeto binario è possibile utilizzare i valori booleani vero e falso ($\{\top, \bot\}$).

Su questa struttura è possibile effettuare due operazioni:

- **rank**: restituisce il numero di elementi uguali a q che sono presenti nella struttura dati fino a x.

$$rank_q(x) = \sum_{k=1}^{k \leq i} B[k], \ \forall i \text{ tale che } 1 \leq i \leq n \ \wedge \ B[k] = q \tag{3.3}$$

- **select**: restituisce la posizione della x-esima occorrenza di q.

$$select_q(x) = \min \{k \in [0, \ldots, n) : \ rank_q(k) = x\} \tag{3.4}$$

Nota 3.

$$rank_q(select_q(i)) = i \tag{3.5}$$

È possibile ottenere una struttura dati succinta, usando $o(n)$ bit aggiuntivi, che permetta di effettuare le operazioni di *rank* e *select* in tempo costante $\mathcal{O}(1)$.

3.1.1 Funzione rank

Vediamo ora come è possibile rendere il tempo di esecuzione dell'operazione di rank costante.

Memorizzare tutti i valori di $rank(i)$ necessiterebbe di $O(n \log n)$ bit il che non lo rende un o-piccolo di n. Memorizzo quindi ogni $l-$esimo valore $rank(i)$, a questo punto, a query time scorriamo i restanti $l-1$ bit.

Questi valori vengono salvati in un vettore *first* $F[0 \dots n/l]$, dove / indica la divisione intera, tale che:

- $F[0] = 0$
- $F[i/l] = rank(i)$ se $i \mod l = 0$

Se $l = \left(\left\lceil \frac{\log n}{2} \right\rceil\right)^2$ si ha un ordine di $\frac{n}{\log n}$ bit per l'array F. Con questa prima operazione è possibile eseguire una rank come:

$$rank(i) = F[i/l] + C(B[l \cdot (i/l) + 1 \dots i]) \tag{3.6}$$

con $C(B')$ che rappresenta una funzione che conta i simboli $\sigma = 1$ in B'. Questa soluzione mi porta a una rank eseguita in tempo $\mathcal{O}(\log^2 n)$.

Possiamo ridurre ulteriormente questo tempo tenendo in memoria più informazioni.

In ogni blocco di lunghezza l indotto da F memorizziamo ogni k posizioni il numero di simboli $\sigma = 1$ a partire dall'inizio del blocco, escludendo la posizione iniziale in cui il valore di rank è già in F. Otteniamo un vettore *second* $S[0 \dots l/k]$ con:

- $S[i/k] = 0$ se $k \mod l = 0$
- $S[i/k] = rank_{B[l\cdot(i/l)+1\dots i]}(i - l \cdot (i/l) + 1)$ se $i \mod k = 0$

Questa soluzione richiede $\mathcal{O}\left(\left(\frac{n}{k}\right) \log l\right)$ bit aggiuntivi. In particolare, scegliendo $k = \left\lceil \frac{\log n}{2} \right\rceil$ si ottiene uno spazio di $\mathcal{O}(\frac{n \log\log n}{\log n})$ bit. Introducendo questo secondo vettore è possibile eseguire una rank come:

$$rank(i) = F[i/l] + S[i/k] + C(B[k \cdot (i/k) + 1 \dots i]) \tag{3.7}$$

Questa soluzione mi porta a una rank eseguita in tempo $\mathcal{O}(\log n)$.

Per ottenere una rank in tempo costante si utilizza la tecnica **Four Russians technique**. Questa tecnica consiste nel salvare una look-up table third T, di dimensioni $2^{k-1} \times k - 1$. T salva i valori di $rank(i')$ per tutte le posizioni $k-1$ in tutte le possibili configurazioni indotte dai blocchi definiti per S. T richiede uno spazio pari a $\mathcal{O}(\sqrt{n} \log n \log\log n)$ bit.

Si definisce $c_i = B[k\cdot(i/k)+1 \dots k\cdot(i/k+1)-1]$ come il bitvector di lunghezza $k-1$ che copre il $(k+1)-$esimo blocco. Usando l'operatore modulo posso sapere la posizione relativa di i in questo bitvector e accedere alla lookup table dove la riga è indicizzata da c_i e la colonna dalla posizione relativa in tempo $\mathcal{O}(1)$.

$$rank(B) = \begin{cases} F[i/l] & \text{se } i \mod l = 0 \\ F[i/l] + S[i/k] & \text{se } i \mod l \neq 0 \wedge i \mod k = 0 \\ F[i/l] + S[i/k] + T[c_i][(i \mod k) - 1] & \text{se } i \mod l \neq 0 \wedge i \mod k \neq 0 \end{cases} \tag{3.8}$$

In questo modo, memorizzando un $o(n)$ bit in più posso eseguire l'operazione di rank in tempo costante. La costruzione della struttura avviene però in tempo lineare.

Esempio 7. *Vediamo un esempio di come può essere implementata la funzione rank in modo da effettuare accesso in tempo costante. Scegliamo i valori di $l = 9$ e $k = 3$ e otteniamo:*

Osservazione 4. *Si può dimostrare che, con un procedimento abbastanza analogo a quello visto per la funzione di rank, è possibile costruire una struttura che necessita di $o(n)$ bit aggiuntivi e che permette di effettuare l'operazione di select in tempo costante.*

3.2 Level Order Unary Degree Sequence

Consideriamo in memoria la rappresentazione di un albero etichettato con n nodi. Una rappresentazione a pointer richiede $\mathcal{O}(n \log n)$ spazio.

Analizziamo ora una rappresentazione basata sulla visita in left-to-rigth level-order di un albero, ovvero una visita in ampiezza. In questa rappresentazione si considera un nodo detto **super-root**. A questo punto si memorizza la sequenza D dei gradi dei nodi. Questa sequenza viene memorizzata utilizzando un prefix

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	···
B		**1**	0	0	**1**	0	**1**	0	**1**	0	**1**	0	0	**1**	0	···
F	0	-	-	-	-	-	-	-	-	4	-	-	-	-	-	···
S	0	-	-	1	-	-	3	-	-	0	-	-	1	-	-	···

T	0	1
00	0	0
01	0	1
10	1	1
11	1	2

Figura 3.1: Esempio di implementazione della funzine rank

code binario, dove per ogni nodo visitato si aggiunge a un bitvector B i simboli della sequenza 1^d0, con d che rappresenta il grado del nodo. La super-root viene aggiunta in modo che il numero di 1 presenti nella sequenza sia uguale al numero di nodi dell'albero.

In questo modo si arriva a ottenere un bitvector di lunghezza $2n+1$ dove si ha un simbolo 1 associato a ogni nodo dell'albero. Con questa rappresentazione posso indicare il nodo m con l'indice relativo del corrispondente bit 1.

Utilizzando le operazioni rank e select definite per il bitvector, è possibile implementare un set di operazioni per interrogare questa rappresentazione dell'albero.

- $is\_leaf(v) = T$ se e solo se $select_0(v) = select_0(v+1)-1$ in quanto per costruzione una foglia aggiunge solo uno 0 al bitvector B, quindi con $select_0(v)$ troviamo la posizione dello 0 posto in B dal nodo antecedente a v nella visita (quello antecedente perché abbiamo il 10 della super-root) e con $select_0(v+1)$ la posizione dello 0 relativo a v. Se sono in due posizioni adiacenti significa che v ha aggiunto solo uno 0 e quindi è una foglia.
- $first\_child(v) = rank_1(select_0(rank_1(select_1(v))) + 1) = rank_1(select_0(v) + 1)$:
  - $k = select_0(v) + 1$ restituisce la posizione k su B del primo "figlio" di v. In altri termini il v-esimo 0 mi dice che ho finito di "visitare" la sotto-sequenza di bitvector costruita per il nodo $v - 1$ e al bit successivo inizia la sequenza del bitvector per v.
  - $m = rank_1(k)$ restituisce il numero di nodo dell'albero in posizione k di B, quindi la label del primo "figlio" di v.
  - Imponiamo che $first\_child(v) = -1$ se $is\_leaf(v) = T$
- $last\_child(v) = rank_1(select_0(rank_1(select_1(v)) + 1) - 1) = rank_1(select_0(v + 1) - 1)$:
  - $k = select_0(v + 1)$ restituisce la posizione k su B dello 0 inserito in visita level-order del nodo con label v. In altri termini, il (v + 1)-esimo 0 mi dice che ho finito di "visitare" la sotto-sequenza di bitvector costruita per il nodo v.
  - $w = k - 1$ restituisce l'indice dell'ultimo 1 inserito in visita level-order del nodo con label j, quindi l'indice su B dell'ultimo "figlio" di c. In altri termini, con la precedente operazione si raggiunge lo 0 di 1^d0 e col -1 l'ultimo 1 di 1^d
  - $m = rank_1(w)$ restituisce il numero di nodo dell'albero in posizione w di B, quindi la label dell'ultimo "figlio" di v.
  - Imponiamo che $last\_child(v) = -1$ se $is\_leaf(v) = T$
- $parent(v) = rank_1(select_1(rank_0(select_1(v)))) = rank_0(select_1(v))$:
  - $i = select_1(v)$ restituisce la posizione i del nodo v nel bitvector B (identificando in quale 1^d0 è stato aggiunto).
  - $j = rank_0(i)$ restituisce il numero di sequenze che sono state aggiunte al bitvector B fino a quella relativa al "genitore" del nodo in posizione i. Il numero di tali sequenza è l'indice del nodo "genitore" per definizione di vista level order e conseguente etichettatura dei nodi.
  - Imponiamo che $parent(v) = -1$ se $v = 1$ (non considero la super-root)
- $degree(v) = last\_child(v) - first\_child(v) + 1$, imponiamo $degree(v) = 0$ se $last\_child(v) = first\_child(v) = -1$

- $nth\_child(v, nth) = rank_1(select_1(first\_child(v)) + nth - 1)$, imponiamo $nth\_child(v, nth) = -1$ se $degree(v) < nth$

Esempio 8. *Costruzione di un e Level Order Unary Degree Sequence:*

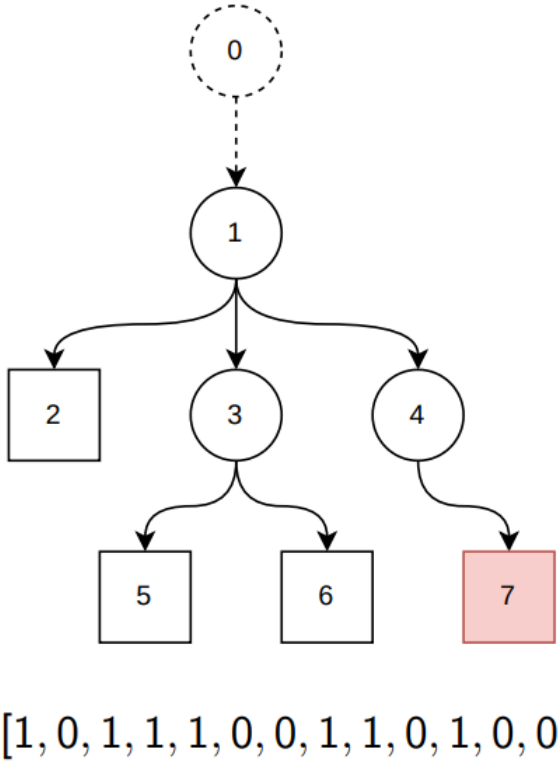

Figura 3.2: Esempio di costruzione di un e Level Order Unary Degree Sequence

3.3 Wavelet Tree

Abbiamo visto come rank e select possono essere usati per interrogare un bitvector, che ha un alfabeto fisso di grandezza 2. Siamo ora interessati a generalizzare tali query ad un alfabeto di grandezza arbitraria. Per praticità assumiamo $\Sigma = [1 \dots s]$ ordinato: $\Sigma[i] \prec \Sigma[j] \iff i < j \dots$.

Dato un generico alfabeto Σ e una sequenza $T[1 \dots n] \in \Sigma$:

- $rank_{\sigma,T}(i)$ conta tutte le occorrenze del simbolo $\sigma \in \Sigma$ fino all'indice i in T, $i \leq |T|$.
- $select_{\sigma,T}(i)$ ritorna la posizione dell'i-esima occorrenza del simbolo $\sigma \in \Sigma$ in T, $i \leq |T|$.

$$rank_{\sigma,T}(select_{\sigma,T}(i)) = i, \ \forall \sigma \in \Sigma \wedge \forall i = 1 \dots n \tag{3.9}$$

Una rappresentazione "naïve" consiste nel considerare come rappresentazione di T un insieme di $|\Sigma| = s$ stringhe binarie $B_\sigma[1 \dots n], \ \forall \sigma \in \Sigma$ si ha che:

$$B_\sigma[i] = 1 \iff T[i] = \sigma \text{ e } B_\sigma[i] = 0 \iff T[i] \neq \sigma \tag{3.10}$$

Se per ogni bitvector B_σ abbiamo calcolato la struttura a supporto vista precedentemente si ha $rank_{\sigma,T}(i)$ in tempo costante $\mathcal{O}(1)$ e uno spazio pari a $s \cdot (n + o(n))$ bit aggiuntivi.

Possiamo ottenere una rappresentazione più efficiente in memoria senza sacrificare troppo i tempi di query. Per realizzare questa rappresentazione consideriamo un **albero binario perfettamente bilanciato** dove ogni nodo corrisponde ad un sottoinsieme di Σ.

I due figli di ogni nodo partizionano il corrispondente sottoinsieme di Σ in due. A ogni nodo v corrisponde una sequenza chiamata R_v (sotto-sequenza dell'input T), la quale è sotto-sequenza di quella con cui è etichettato il nodo genitore di v. Alla root corrisponde la sequenza $R_v = T$.

A ogni nodo v si associa un bitvector, denotato con B_v, che indica a quale dei due figli del nodo v ogni simbolo della sotto-sequenza R_v appartiene. Ad esempio con $1 \leq j \leq |R_v|$ se $B_v[j] = 0$ allora il carattere associato $R_v[j]$ appartiene alla sotto-sequenza rappresentata dal figlio di sinistra, mentre se $B_v[j] = 1$ allora il carattere associato a $R_v[j]$ appartiene alla sotto-sequenza rappresentata dal figlio di destra.

Le foglie dell'albero sono *virtualmente* etichettate con i singoli caratteri dell'alfabeto. In realtà ci basta la funzione $rank_1$ eseguita sui bitvector che etichettano i genitori delle foglie per recuperare l'informazione.

Vediamo ora come realizzare le operazioni su questa struttura dati:

- **rank**: si inizia il calcolo nel nodo root e si determina a quale dei due figli appartiene σ, tramite l'ordinamento dell'alfabeto. Nella radice se $B_v[j] = 0$ allora $R_v[j] = \sum \left[\left\lceil \frac{s}{2} \right\rceil\right] \vee R_v[j] \prec \Sigma \left[\left\lceil \frac{s}{2} \right\rceil\right]$, altrimenti se $B_v[j] = 1$ allora il carattere che si cerca è nella seconda metà dell'alfabeto.

  A questo punto si prosegue nel seguente modo:

  - se $\sigma = \Sigma \left[\left\lceil \frac{s}{2} \right\rceil\right] \vee \sigma \prec \Sigma \left[\left\lceil \frac{s}{2} \right\rceil\right]$ allora si prosegue verso il "figlio" di sinistra, aggiornando i con $i = rank_{0,B_v}(i)$.
  - Altrimenti usiamo il "figlio" di destra, aggiornando i con $i = rank_{1,B_v}(i)$.

  Nel nuovo v si procede nello stesso modo, considerando che ora $\Sigma = \Sigma \left[1 \ldots \left\lceil \frac{s}{2} \right\rceil\right]$ se si è andati a sinistra e $\Sigma = \Sigma \left[\left\lceil \frac{s}{2} \right\rceil + 1 \ldots s\right]$ se a destra.

  Si prosegue fino ad una foglia e $rank_{\sigma,T}(i) = i'$, dove i' è l'ultimo valore di i che si ottiene nei vari step.

  L'albero ha altezza $\lceil \log s \rceil$ quindi $rank_{\sigma,T}(i)$ può essere calcolato in $\mathcal{O}(\log s)$, dove s è la cardinalità dell'alfabeto.

- **random access**: in questo caso la scelta del figlio dipende unicamente da $B_v[i]$:

  - Se $B_v[i] = 0$ proseguo verso il "figlio" di sinistra, con $i = rank_{0,B_v}(i)$
  - Se $B_v[i] = 1$ proseguo verso il "figlio" di destra, con $i = rank_{1,B_v}(i)$

  $T[i]$ è il simbolo che etichetta la foglia raggiunta alla fine della visita dato che il wavelet tree di una sequenza T garantisce random access alla sequenza stessa possiamo sostituirla col suo wavelet tree.

  L'albero ha altezza $\lceil \log s \rceil$ quindi $access_{\sigma,T}(i)$ può essere calcolato in $\mathcal{O}(\log s)$.

- **select**: analogamente ai bitvector si può dimostrare che anche $select_{\sigma,T}(i)$ può essere calcolato in $\mathcal{O}(\log s)$

Vediamo ora come costruire un wavelet tree livello per livello a partire dalla root:

1. Si etichetta la root con $R_v = T$ e un bitvector B_v tale che $\forall 1 \leq i \leq |T|$:
   - $B_v[i] = 0 \iff T[i] = \Sigma \left[\left\lceil \frac{s}{2} \right\rceil\right] \vee T[i] \prec \Sigma \left[\left\lceil \frac{s}{2} \right\rceil\right]$
   - $B_v[i] = 1$ altrimenti.
2. Si estrae la sotto-sequenza T' corrispondente ai valori 0 di B_v e la si usa per etichettare il "figlio" di sinistra v_1 (che è relativo all'alfabeto $\Sigma = \Sigma \left[1 \ldots \left\lceil \frac{s}{2} \right\rceil\right]$) mentre quella corrispondente ai valori 1 la si usa per etichettare il "figlio" di destra v_2 (che è relativo all'alfabeto $\Sigma = \Sigma \left[\left\lceil \frac{s}{2} \right\rceil + 1 \ldots s\right]$) per entrambe.
3. Posso quindi cancellare T e costruire i bitvector B_{v1} e B_{v2}, con le rispettive strutture per la funzione rank e continuare ricorsivamente fino al raggiungimento delle foglie (quando si raggiungono alfabeti di cardinalità 1).

Il processo di costruzione di un wavelet tree richiede un tempo pari a $\mathcal{O}(n \log s)$.

In ogni momento della costruzione del wavelet tree abbiamo un bound in spazio pari a $3n \log s + \mathcal{O}(s \log n)$, dato da:

- 3 sottosequenze di T, quella del "genitore" e quelle dei due "figli".
- Tutti i bitvector finora computati che formano il wavelet tree.
- I puntatori che memorizzano la struttura ad albero.

Un ulteriore miglioramento in spazio si può ottenere concatenando tutti i bitvector in un unico bitvector con una sola struttura a supporto della funzione rank. In tal caso la struttura ad albero si può inferire dai partizionamenti dell'alfabeto e dai bitvector. Questa variante è chiamata **levelwise wavelet tree**.

Un wavelet tree per una sequenza lunga n costruita su alfabeto di cardinalità s occupa, avendo $\mathcal{O}(s \log n)$ per la topologia dell'albero:

$$n \log s + o(n \log s) + \mathcal{O}(s \log n) \; bit \tag{3.11}$$

Mentre, un levelwise wavelet tree richiede:

$$n \log s + o(n \log s) \; bit \tag{3.12}$$

Esempio 9. *Costruzione di un wavelet tree:*

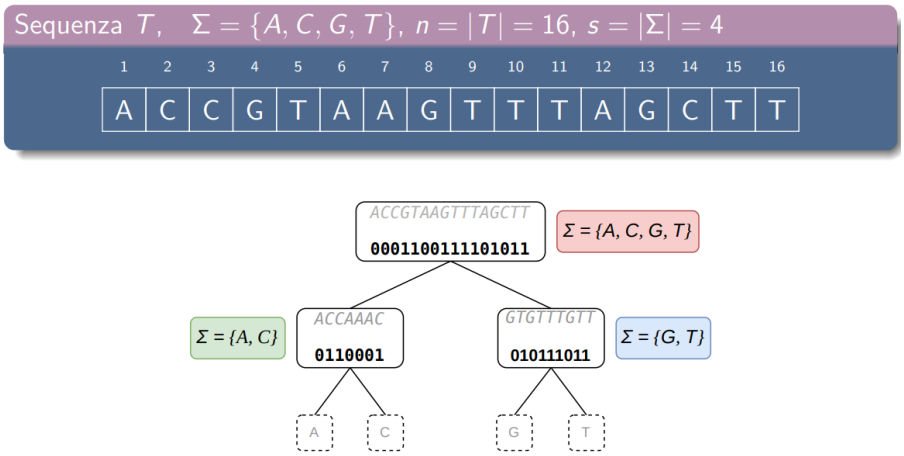

Figura 3.3: Esempio di costruzione di un wavelet tree su una sequenza semplice

3.4 Altre strutture dati succinte

- **Parentesi bilanciate**: si costruisce a partire dalla DFS dell'albero (preorder):
  - "(" quando si raggiunge un nodo per la prima volta.
  - ")" quando si è terminata la visita del sottoalbero.
- Strutture dati succinte che supportano le **range minimum queries**.
- **Wavelet matrix**: nascono con l'idea di migliorare i levelwise wavelet tree nella gestione di larghi alfabeti:
  - tempi di access dimezzati
  - tempi di rank e select leggermente ridotti

  L'idea è che i bit di un nodo "figlio" non sono più "allineati" al "genitore" ma si assume che passando da un livello all'altro, tutti gli zero vanno da una parte e gli uni dall'altra.

  Salvando ad ogni livello l il numero totale di simboli $\sigma = 0$ z_l, richiedendo in totale $\mathcal{O}(\log n \log s)$ bit, si ottiene lo stesso comportamento di un levelwise wavelet tree.

  Una wavelet matrix richiede $n \log s + o(n \log s)$ bit, può essere costruita in $\mathcal{O}(n \log s)$ e risponde alle stesse query di un (levelwise) wavelet tree in $\mathcal{O}(\log s)$

Definizione 22 (**Range Minimum Query**). *Dato un array $A[1 \dots n]$ di numeri n elementi da un universo totalmente ordinato la **Range Minimum Query** $RMQ_A(i, j)$, con $1 \leq i \leq j \leq n$, restituisce la posizione k di un elemento minimo in $A[i \dots j]$:*

$$RMQ_A(i, j) = argmin_{i \leq k \leq j}\{A[k]\}$$

Si può dimostrare che è possibile costruire una struttura dati succinta che richiede $2n + o(n)$ bit in memoria e che risponde in $\mathcal{O}(1)$.

Capitolo 4

Strutture Dati Probabilistiche Hashing-Based

Su strutture dati semplici come array matrici... posso eseguire operazioni di accesso, cancellazione di un elemento e inserimento di un elemento in tempo $\mathcal{O}(1)$ data la chiave k, l'input x e la posizione $k[x]$.

Con queste strutture dati si ha un problema in quanto è possibile che l'insieme delle chiavi utilizzate sia molto minore rispetto alla dimensione della struttura dati. Una soluzione a questo problema sono le **hash tables**.

Se con l'accesso diretto l'elemento con chiave k era memorizzato in posizione k, con l'hash è memorizzato in $h(k)$, dove h è una funzione di hash.

Definizione 23 (**Funzione di hash**). *Una **funzione di hash** h è definita come:*

$$h : \mathcal{U} \to \{0, \dots, m-1\} \tag{4.1}$$

ovvero come una funzione definita dall'insieme universo $\mathcal{U}$ all'insieme delle posizioni $\{0, \dots, m-1\}$ di una tabella di hash T.

Le funzioni di hash possono avere input "*scomodi*", ovvero input che possono generare **collisioni**:

$$h(k') = h(k'') \quad \text{con} \quad k' \neq k'' \tag{4.2}$$

Una possibile soluzione per ridurre le collisioni consiste nell'utilizzare una famiglia di funzioni di hash al posto di una singola.

Definizione 24 (**famiglia di funzioni hash**). *Una **famiglia di funzioni hash** è un insieme $\mathcal{H}$ di funzioni hash con lo stesso dominio e codominio. La scelta di $h \in \mathcal{H}$ può essere fatta con un sampling uniforme su $\mathcal{H}$.*

$\mathcal{H}$ è detta **universale** se e solo se, con $h : \mathcal{U} \to \{0, \dots, m-1\}$ scelta casualmente da $\mathcal{H}$, si ha che:

$$\mathcal{P}(h(x) = h(y)) \leq \frac{1}{m} \tag{4.3}$$

ovvero la probabilità di collisioni è minore di $\frac{1}{m}$ dove m è la dimensione della tabella.

Un'altra possibile soluzione consiste nelle hash table dove le collisioni sono risolte tramite concatenazione. Si mettono tutti gli elementi che collidono nella stessa posizione dell'hash table in una lista concatenata. Chiamiamo α il **fattore di carico**, ovvero il numero medio di elementi in queste liste concatenate.

Il caso peggiore si ha quando tutte le n chiavi "mappano" in una sola lista, quindi i tempi di accesso richiedonoun tempo paria a $\theta(n)$ ma nella realtà è difficile che accada quindi accesso in $\theta(1 + \alpha)$ nel caso migliore, con il numero di posizioni nella hash table proporzionale al numero di elementi della tabella (quindi $\alpha \to 1$), ho accesso in tempo $\theta(1)$.

4.1 Membership problem

Definizione 25 (**Membership problem**). *Il problema **membership problem** è definito come:*

- ***Input**:*

  - *Insieme universo $\mathcal{U}$, $|\mathcal{U}| = u$ che per praticità assumiamo valori interi con ogni elemento che occupa $w = \log u$ bit.*
  - *Insieme $S \subseteq \mathcal{U}$, $|S| = n$.*
  - *Un elemento $y \in \mathcal{U}$.*
- ***Output**: T se $y \in S$, F altrimenti.*

Una prima soluzione per questo problema consiste nel creare una hash table per S con le collisioni risolte tramite liste concatenate. Questa struttura ci permette di ottenere una risposta in tempo pari a $\theta(1)$ occupando uno spazio pari a $\mathcal{O}(n \log u)$ bit.

È possibile ottenere una soluzione migliore se assumiamo di poter ammettere falsi positivi ma comunque non falsi negativi. Nel caso in cui ammettiamo falsi positivi ma non falsi negativi parliamo del problema di **approximate membership**.

Se $y \in S$ voglio sempre ottenere T, quindi ho sempre l'informazione corretta in merito al fatto che un elemento y sia in S. Se $y \notin S$ voglio ottenere F con probabilità $\mathcal{P} \geq 1 - \delta$, con $\delta \in \mathbb{R}^+ \wedge \delta \to 0$. Si assume quindi di avere errori sui falsi positivi, ovvero ottengo T e non F con probabilità $\mathcal{P} \leq \delta$

Creiamo una struttura con $\frac{n}{\delta}$ bit, per S, $|S| = n$ insieme universo $\mathcal{U}$. Sia $\mathcal{H}$ una famiglia universale di funzioni hash per $\mathcal{U}$ con $h_j : \mathcal{U} \to \{0, \ldots, m-1\}$, con $m = \frac{n}{\delta}$. Prendendo casualmente una funzione di hash $h \in \mathcal{H}$, popoliamo un bitvector A, $|A| = m = \frac{n}{\delta}$, tale che:

$$A[i] = \begin{cases} 1 & \text{se } \exists k \in S \text{ tale che } h(k) = i \\ 0 & \text{altrimenti} \end{cases} \tag{4.4}$$

Sulla struttura dati appena creata è possibile eseguire le query per sapere se $x \in S$ in tempo $\mathcal{O}(1)$ ($A[h(x)] = 1$).

- Se $x \in S$ si ottiene sempre T.
- Se $x \notin S$ si ottiene lo stesso T se e solo se $\exists k \in S$ tale che $h(k) = h(x)$.
- $\mathcal{H}$ famiglia universale quindi $\mathcal{P}[h(k) = h(x)] \leq \frac{1}{m} = \frac{\delta}{n}$
- La probabilità che esista almeno una tale chiave k è ($\mathcal{P}(A \cup B) \leq \mathcal{P}(A) + \mathcal{P}(B)$):

$$\sum_{k \in S} \mathcal{P}[(h(k) = h(x))] \leq \frac{n}{m} = \frac{(\delta \cdot n)}{n} = \delta \tag{4.5}$$

4.2 Bloom Filter

Definizione 26 (*Risultato solo teorico*)**.** *Data una funzione di hash $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$, per praticità $\mathcal{U} \subseteq \mathbb{N}$, questa è **ideale** se e solo se, $\forall k \in \mathcal{U}$, $h(k)$ vale indipendentemente un valore uniformemente distribuito su $[0 \ldots m-1]$. Quindi, $\forall k \in \mathcal{U}$, $h(k)$ vale un qualsiasi intero tra 0 e $m-1$ con la stessa probabilità e tale valore non dipende dal valore di hash delle altre chiavi.*

Struttura dati per S, $|S| = n$. Si considera un bitvector A, $|A| = m$, inoltre, si considera una famiglia di l funzioni hash ideali $\mathcal{H} = \{h_0, \ldots, h_{l-1}\}$ dove:

$$h : \mathcal{U} \to \{0, \ldots, m-1\}, \ \forall h \in \mathcal{H} \tag{4.6}$$

Il bitvector A viene riempito nel seguente modo:

$$\forall k \in S \text{ e } \forall h_i \in \mathcal{H} : A[h_i(k)] = 1 \tag{4.7}$$

quindi per ogni $k \in S$ abbiamo fino a l bit pari a 1 in A. Si utilizza il termine "fino a" perché alcune $h_i, h_j \in \mathcal{H}$ potrei avere $h_i(k) = h_j(k)$ e se già $A[h_i(k)] = 1$ non avrò ulteriori modifiche in posizione $h_i(k) = h_j(k)$.

Il bitvector A è denominato **Bloom filter** di S.

Su questa struttura appena creata è possibile eseguire le query per l'approximate membership problem come dato $x \in \mathcal{U}$ si ha che $x \in S$ se e solo se:

$$A[h_i(x)] = 1, \ \forall h_i \in \mathcal{H} \tag{4.8}$$

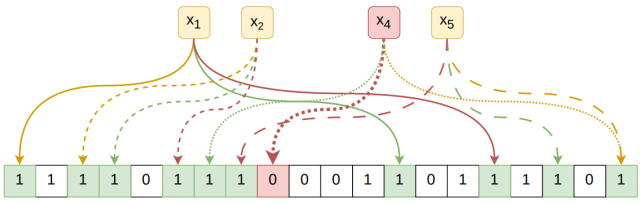

Figura 4.1: Esempio di query su Bloom Filter

Una generalizzazione dei Bloom filter sono **Counting Bloom filter**. In queste strutture dati si tiene conto anche di quante funzioni di hash mappano in una certa posizione dato un elemento qualsiasi si verifica la presenza tramite il counting Bloom filter tramite una threshold θ.

Possiamo costruire queste strutture dati andando in fase di preprocessing $\forall k \in S$ e $\forall h_i \in \mathcal{H} : A[h_i(k)] += 1$.
In fase di query, dato $x \in \mathcal{U}$ abbiamo:

- Se $\exists h_i \in \mathcal{H}$ tale che $A[h_i(x)] = 0$ o $A[h_i(x)] \leq \theta$ allora $x \notin S$
- Se $\forall h_i \in \mathcal{H}$ $A[h_i(x)] > 0$ o $A[h_i(x)] > \theta$ allora "probabilmente" $x \in S$, avendo $\mathcal{P}(FP) \neq 0$
- $A[h_i(x)]$ è una sovrastima del numero di elementi x in S.

4.3 Heavy Hitters Problem

L'**heavy hitters problem** consiste nell'identificazione dell'elemento più frequente o anche detto **heavy hitter**. Questo problema non può essere risolto utilizzando un **Bloom Filter** in quanto richiederebbe troppo spazio di memorizzazione.

Definizione 27 (**Stream di dati**). *Con **stream di dati** si intende una sequenza di dati passati uno ad uno alla struttura dati. Quindi, data una sequenza $S = s_0, s_1, \ldots, s_{n-1}$, prima si considera s_0, poi s_1 ... fino a s_{n-1} si costruisce quindi una struttura dati che può essere interrogata con nuovi valori $x \in \mathcal{U}$.*

Su questa struttura dati è possibile effettuare le seguenti query:

- Quante volte appare x nello stream:

$$|\{i \in \{0, 1, \ldots, n-1\} | \ s_i = x\}| \tag{4.9}$$

- Quanti elementi distinti si hanno nello stream:

$$|\{s_i | i \in \{0, 1, \ldots, n-1\}\}| \tag{4.10}$$

4.3.1 Count-min sketch

Una struttura dati che ci permette di risolvere questo problema è il **Count-min sketch**, la quale richiede poco spazio in memoria ed è concettualmente simile a un Bloom filter.

Questa struttura dati è definita a partire da:

- Insieme universo $\mathcal{U}$.
- Uno stream S lungo n costruito da $\mathcal{U}$.
- Due parametri d'errore δ e ε. Otterremo risposte alle query "sbagliate" entro un fattore aggiuntivo ε con probabilità almeno $1 - \delta$
- $\mathcal{H}$, $|\mathcal{H}| = l = \lceil \ln \frac{1}{\delta} \rceil$, famiglia di funzioni hash universale per $\mathcal{U}$ si impone che:

$$h_i : \mathcal{U} \to \{0, \ldots, m\}, \ \forall h_i \in \mathcal{H}, \ \text{con } m = \left\lceil \frac{e}{\varepsilon} - 1 \right\rceil \tag{4.11}$$

Il **Count-min sketch** è costituito da una matrice bidimensionale T con l righe, una per ogni $h_i \in \mathcal{H}$, e m colonne. Si hanno quindi l hash table indipendenti con m entry ciascuna. La matrice T viene inizializzata con tutti gli elementi a 0.

Il caricamento di T avviene nel seguente modo:

- Si considerano in ordine tutti gli $x_j \in S$, con $j = 0, 1, \ldots, n-1$.
- Sappiamo che ogni h_i ha di fatto come codominio l'insieme degli indici di colonna. Quindi inserire x_j in T vuol dire incrementare di 1 $T_{h_i}[h_i(x_j)]$, $\forall h_i \in H$

Queste operazioni richiedono un tempo per essere eseguite pari a $\mathcal{O}(n)$, dove n è la lunghezza dello stream.

Su questa struttura dati è possibile eseguire la query per $q \in \mathcal{U}$ nel seguente modo:

- Si applica ogni funzione di hash a q
- Si tiene traccia di ogni $T_{h_i}[h_i(q)]$, $\forall h_i \in \mathcal{H}$
- Si restituisce il minimo tra tali valori, che è una stima (una frequenza approssimata $\hat{a}_q$) di quante volte occorre q in S:

$$\hat{a}_q = \min_{h_i \in \mathcal{H}} T_{h_i}[h_i(q)] \tag{4.12}$$

Una query si effettua in tempo $\mathcal{O}(l)$.

È possibile dimostrare che data a_q, ovvero la frequenza reale di q in S, si ha che:

$$a_q \leq \hat{a}_q \tag{4.13}$$

ù Si dimostra che $a_q \leq \hat{a}_q$ a causa delle collisioni si ottengono sovrastime ma mai sottostime della frequenza.

Inoltre, è possibile dimostrare:

$$\hat{a}_q \leq a_q + \varepsilon \cdot n \tag{4.14}$$

con probabilità almeno $1 - \delta$.

Dato che la matrice bidimensionale T è di dimensione $l \times m = \lceil \ln \frac{1}{\delta} \rceil \times \lceil \frac{e}{\varepsilon} \rceil$ con valori che richiedono $\log n$ bit, allora la struttura dati occupa uno spazio pari a:

$$\left(\left\lceil \ln \frac{1}{\delta} \right\rceil \cdot \left\lceil \frac{e}{\varepsilon} \right\rceil \cdot \log n \right) \text{ bit} \tag{4.15}$$

$\mathcal{S} = \{4, 2, 6, 2, 2, 7, 1, 6\}$

T	ξ_1	ξ_2	ξ_3	ξ_4	ξ_5	ξ_6	ξ_7	ξ_8	ξ_9	ξ_{10}	ξ_{10}
h_1	0	0	0	0	0	0	0	0	0	0	0
h_2	0	0	0	0	0	0	0	0	0	0	0
h_3	0	0	0	0	0	0	0	0	0	0	0
h_4	0	0	0	0	0	0	0	0	0	0	0

(a) Inizializzazione

$\mathcal{S} = \{4, 2, 6, 2, 2, 7, 1, 6\}$

$h_1(4) = \xi_2 \quad h_2(4) = \xi_1 \quad h_3(4) = \xi_7 \quad h_4(4) = \xi_{11}$

T	ξ_1	ξ_2	ξ_3	ξ_4	ξ_5	ξ_6	ξ_7	ξ_8	ξ_9	ξ_{10}	ξ_{10}
h_1	0	**1**	0	0	0	0	0	0	0	0	0
h_2	**1**	0	0	0	0	0	0	0	0	0	0
h_3	0	0	0	**1**	0	0	0	0	0	0	0
h_4	0	0	0	0	0	0	0	0	0	0	**1**

(b) Inserimento del primo elemento

$\mathcal{S} = \{4, 2, 6, 2, 2, 7, 1, 6\}$

$h_1(2) = \xi_6 \quad h_2(2) = \xi_2 \quad h_3(2) = \xi_8 \quad h_4(2) = \xi_1$

T	ξ_1	ξ_2	ξ_3	ξ_4	ξ_5	ξ_6	ξ_7	ξ_8	ξ_9	ξ_{10}	ξ_{10}
h_1	0	1	0	0	0	**1**	0	0	0	0	0
h_2	1	**1**	0	0	0	0	0	0	0	0	0
h_3	0	0	0	1	0	0	0	**1**	0	0	0
h_4	**1**	0	0	0	0	0	0	0	0	0	1

(c) Inserimento del secondo elemento

$\mathcal{S} = \{4, 2, 6, 2, 2, 7, 1, 6\}$

$h_1(6) = \xi_8 \quad h_2(6) = \xi_{10} \quad h_3(6) = \xi_5 \quad h_4(6) = \xi_9$

T	ξ_1	ξ_2	ξ_3	ξ_4	ξ_5	ξ_6	ξ_7	ξ_8	ξ_9	ξ_{10}	ξ_{10}
h_1	0	1	0	0	0	1	0	**1**	0	0	0
h_2	1	1	0	0	0	0	0	0	0	**1**	0
h_3	0	0	0	1	**1**	0	0	1	0	0	0
h_4	1	0	0	0	0	0	0	0	**1**	0	1

(d) Inserimento del terzo elemento

Figura 4.2: Esempio di riempimento Count-min sketch

Capitolo 5

Pattern Matching su Stringhe

5.1 Introduzione

Il problema del Pattern Matching consiste cercare un motivo all'interno di un oggetto più o meno complesso. In questo corso ci si concentrerà sul Pattern Matching su Stringhe, ovvero cercare all'interno di un testo T le occorrenze di un pattern P.

Definizione 28 (**Stringa**). *Definiamo una **stringa** X come una giustapposizione di simboli appartenenti a un alfabeto Σ.*

$$X = x_1 x_2 \dots x_n \quad x_i \in \Sigma \ \forall i = 1, \dots, n \tag{5.1}$$

In aggiunta, definiremo:

- ***Stringa nulla** ε è una stringa composta da zero simboli.*
- *Simbolo in posizione i si riferisce al simbolo in posizione i-esima $x_i = X[i]$*
- ***Sottostringa** da i a j è una porzione di stringa compresa tra gli indici i e j.*

  $$X[i, j] = X[i : j] = X[i]X[i+1] \dots X[j-1]X[j] \tag{5.2}$$

  Posso esprimerlo attraverso la seguente notazione: $X[i, j] \vee X[i : j]$. Possiamo dire che una sottostringa $X[i, j]$ si definisce:

  - ***Propria** se $i \neq 1 \wedge j \neq |X|$*
  - ***Impropria** altrimenti*

- ***Prefisso** di lunghezza j è una sottostringa $X[1, j]$. Anche in questo caso possiamo distinguere:*

  - ***Proprio** se $j \neq |X|$*
  - ***Improprio** altrimenti*

  Per il prefisso è possibile definire anche il prefisso nullo, ovvero il prefisso composto da zero caratteri ($X[1, j] = \epsilon \ \rightarrow \ j = 0$).

- ***Suffisso** che inizia in i è la sottostringa $X[i, |X|]$. Di questa sottostringa posso calcolare la lunghezza del prefisso come:*

  $$|X[i, |X|]| = |X| - i + 1 \tag{5.3}$$

  Anche in questo caso possiamo distinguere:

  - ***Proprio** se $i \neq 1$*
  - ***Improprio** altrimenti*

  È possibile definire il suffisso nullo ovvero quello composto da zero caratteri ($X[i, |X|] = \epsilon \ \rightarrow \ i = |X|+1$).

Nota 4. *Una **stringa** di caratteri si differenzia da un **sequenza** degli stessi caratteri dal momento che:*

- ***stringa**: è una giustapposizione di caratteri, quindi sono tutti concatenati*

  $$X = \sigma_1 \sigma_2 \dots \sigma_n \tag{5.4}$$

- ***sequenza**: è un elenco di caratteri separati da un separatore*

$$X =< \sigma_1, \sigma_2, \ldots, \sigma_n > \tag{5.5}$$

Quando si parla di string matching possiamo definire due tipi. Dati un pattern P e un testo T possiamo definire lo String Matching:

1. **Esatto**: consiste nel cercare le occorrenze esatte di P in T.
2. **Approssimato**: consiste nel cercare le occorrenze approssimate di P in T.

5.1.1 String Matching Esatto

Possiamo definire il problema di **string matching esatto** formalmente nel seguente modo:

- **input**: un testo T e un pattern P, rispettivamente $|T| = n$ e $|P| = m$, definiti su un alfabeto Σ.
- **output**: tutte le occorrenze esatte i di T ($T[i, i + m - 1] = P$).

Definizione 29 (**Occorrenza esatta**). *Una posizione i del testo T tale che $T[i, i+m-1] = P$ è un'**occorrenza esatta** di P in T.*

Definizione 30 (**Match**). *Dati due simboli $s_1, s_2 \in \Sigma$ si ha un **match** se $s_1 = s_2$.*

Definizione 31 (**Mismatch**). *Dati due simboli $s_1, s_2 \in \Sigma$ si ha un **mismatch** se $s_1 \neq s_2$.*

Avendo definito il problema in questo modo è possibile definire un semplice algoritmo che mi permette di calcolare l'output del problema. Questo algoritmo utilizza una finestra W, delle stesse dimensioni del pattern, che scorre sul testo. L'algoritmo semplice che permette di calcolare le occorrenze esatte è:

1. Uso una finestra W lunga m che scorre lungo T da sinistra a destra. La posizione iniziale di W è $i = 1$.
2. Si confronta ogni simbolo di P con il corrispondente simbolo di T all'interno di W da sinistra verso destra.

$$P[j] = T[i + j - 1] \ \forall \ j \text{ tale che } 1 \leq j \leq m \ \Rightarrow T[i, i + m - 1] = P \tag{5.6}$$

3. W viene spostata di una posizione verso destra e il confronto viene ripetuto.
4. Ultima posizione di W è $i = |T| - |P| + 1 = n - m + 1$

Algorithm 1 Algoritmo banale per String Matching Esatto

```
function TRIVIAL_EXACT_OCCURRENCES(T, P)
    n ← |T|
    m ← |P|
    i ← 1
    while i ≤ n − m + 1 do
        j ← 1
        while P[j] = T[i + j − 1] ∧ j ≤ m do
            j ≤ j + 1
        end while
        if j = m + 1 then
            output i
        end if
        i ← i + 1
    end while
end function
```

Questo algoritmo richiede un tempo pari a $\mathcal{O}(m \cdot n)$.

Nota 5. *Questo algoritmo può essere migliorato spostando la finestra alla posizione successiva al primo mismatch oppure si può effettuare un preprocessing del pattern e del testo, permettendo di passare da una complessità quadratica ad una logaritmica o lineare.*

5.1.2 String Matching Approssimato

Definiremo il problema di **string matching approssimato** formalmente nel seguente modo:

- **input**: testo T e un pattern P, rispettivamente $|T| = n$ e $|P| = m$, definiti entrambi su un alfabeto Σ, infine, una soglia k di errore.
- **output**: tutte le occorrenze approssimate di P in T che respettano la soglia di errore k.

Introducendo una soglia di errore abbiamo bisogno di definire una metrica per calcolarlo. Per fare ciò si utilizza la *distanza di edit* (ED) tra due stringhe. Tale distanza è definita come il minimo numero di operazioni di sostituzione, cancellazione, inserimento di un simbolo che trasformano una stringa nell'altra.

Nota 6.

$$ED(X_1, X_2) \geq abs(|X_1| - |X_2|) \tag{5.7}$$

Definizione 32 (**Occorrenza approssimata**)**.** *Una posizione i del testo T, tale che esista almeno una sottostringa $S = T[i - L + 1, i]$, con $ED(P, S) \leq k$, che è un'occorrenza approssimata di P in T.*

Nota 7. *Si può aggiungere:*

1. *Se $ED(P, S) \leq k$, allora i è occorrenza approssimata.*
2. *$ED(P, S) \geq abs(m - L) \Rightarrow$ se $abs(m - L) > k$ allora i non può essere occorrenza approssimata.*

Quindi il problema formale dello **string matching approssimato** verrà definito come:

- **input**: un testo T e un pattern P, rispettivamente $|T| = n$ e $|P| = m$, definiti entrambi su un alfabeto Σ, infine, una soglia k di errore.
- **output**: tutte le occorrenze approssimate di P in T tale che $ED(P, S) \leq k$.

Avendo definito il problema in questo modo è possibile definire un semplice algoritmo che mi permette di calcolare l'output del problema. Questo algoritmo utilizza una finestra W, di dimensione variabile, che scorre sul testo.

1. Uso una finestra W di lunghezza variabile $\in [m - k, m + k]$, per un totale di $2k + 1$ ampiezze da testare, che scorre lungo il testo T da sinistra a destra. La posizione iniziale di tale finestra è $i = m - k$ e la sua lunghezza iniziale è $m - k$.
2. Se la distanza di edit tra P e la sottostringa di T compresa in W è $\leq\ k$, allora i è occorrenza approssimata di P in T.
3. W viene spostata a destra di una posizione.

Algorithm 2 Algoritmo banale per String Matching Approssimato

```
function TRIVIAL_APPROX_OCCURRENCES(T, P, k)
    n ← |T|
    m ← |P|
    i ← m − k
    while i ≤ n do
        L ← m − k
        while L ≤ m + k ∧ i − L + 1 ≥ 1 do
            if ED(P, T[i − L + 1, i]) ≤ k then
                output i
            end if
            L ← L + 1
        end while
        i ← i + 1
    end while
end function
```

Questo algoritmo mi permette di calcolare il matching approssimato in tempo $\mathcal{O}(n \cdot k \cdot m^2)$, dove m^2 è dovuto al calcolo della distanza di edit tra le due sottostringhe.

5.2 Ricerca esatta con Automa a Stati Finiti

Un automa è un modello di calcolo che riconosce un linguaggio, ovvero un insieme di stringhe che godono di una proprietà. Gli automi a stati finiti riconoscono un linguaggio regolare.

Definizione 33 (**Automa a stati finiti**). *Un Automa a Stati Finiti è formalmente una quintupla:*

$$A = (Q, \Sigma, \delta, q_0, F) \tag{5.8}$$

dove:

- Q*, insieme finito di stati.*
- Σ*, alfabeto in input*
- $\delta : Q \times \Sigma \to Q$*, funzione di transizione.* $\delta(q, \sigma)$ *è lo stato di arrivo a partire da* q *dopo la lettura di* σ
- q_0*, stato iniziale*
- F *(sottoinsieme di* Q*), insieme degli stati accettanti.*

Gli automi a stati finiti possono essere rappresentati attraverso un diagramma di stato, ovvero attraverso una struttura a grafo dove i vertici sono gli stati. Esiste l'arco (q_1, q_2) se almeno un simbolo σ è tale per cui $\delta(q_1, \sigma) = q_2$. L'arco (q_1, q_2) viene etichettato dalla lista di simboli che permettono la transizione da q_1 a q_2. Lo stato iniziale q_0 è indicato tramite un arco entrante che non esce da uno stato, mentre gli stati accettanti sono indicati da un doppio bordo.

È possibile rappresentare la funzione di transizione δ degli automi attraverso una matrice T con $|Q|$ righe e $|\Sigma|$ colonne. Nella generica cella (q, σ) sarà contenuto il valore di $\delta(q, \sigma)$, ovvero lo stato di arrivo a partire dallo stato q attraverso il simbolo σ.

$$T[q, \sigma] = q' \iff \delta(q, \sigma) = q' \tag{5.9}$$

Definizione 34 (**Bordo**). *Il* ***bordo*** *di una stringa* X *è il più lungo prefisso* ***proprio*** *di* X *che occorre come suffisso di* X*.*

Esempio 10. *Esempi di bordo:*

- $X = baaccbbaac$ *il suo bordo sarà* $B(X) = baac$
- $X = aaaccbbaac$ *il suo bordo sarà* $B(X) = \varepsilon$
- $X = abababa$ *il suo bordo sarà* $B(X) = ababa$
- $X = aaaaaaaa$ *il suo bordo sarà* $B(X) = aaaaaaa$
- $X = a$ *il suo bordo sarà* $B(X) = \varepsilon$

Nota 8. *Il bordo di un simbolo è sempre vuoto.*

Definizione 35 (**Concatenazione**). *La* ***concatenazione*** *di un simbolo* σ *con la stringa* X *è la stringa* $X\sigma$*.*

Possiamo definire ora l'automa a stati finiti per la ricerca esatta di un pattern P di lunghezza m definito su alfabeto Σ è la quintupla $(Q, \Sigma, \delta, q_0, F)$ con:

- $Q = \{0, 1, \ldots, m\}$
- Σ è l'alfabeto di definizione di P
- $\delta : Q \times \Sigma \to Q$ è la funzione di transizione
- $q_0 = 0$ è lo stato iniziale
- $F = \{m\}$ è lo stato accettante

A questo punto il processo di ricerca esatta attraverso un automa a stati finiti è composto da:

1. **Preprocessing del pattern**: costruisco l'automa per il pattern P che consiste nel calcolo della funzione di transizione δ in tempo $\theta(m \cdot |\Sigma|)$.
2. **Ricerca nel testo**: uso dell'automa per riconoscere, in un testo T definito su alfabeto Σ, tutte le occorrenze esatte di P. La scansione del testo T avviene in tempo $\theta(n)$

5.2.1 Funzione di transizione

La funzione di transizione δ per un pattern P di lunghezza m definito su un alfabeto Σ è definita per ogni $(j, \sigma) \in Q \times \Sigma$ tale che $\delta(j, \sigma)$ è lo stato in cui si arriva da j attraverso σ:

$$\delta(j,\sigma) = \begin{cases} j+1 & \text{se } j < m \ \wedge \ P[j+1] = \sigma \\ k & \text{se } j = m \ \vee \ P[j+1] \neq \sigma \end{cases} \tag{5.10}$$

dove k è la lunghezza del bordo del prefisso di P di lunghezza $1, j$ a cui è concatenato σ, ovvero:

$$k = |B(P[1,j]\sigma)| \quad k \leq j \tag{5.11}$$

Dallo stato 0 si arriva allo stato 0 per qualsiasi simbolo diverso da P[1]. Dallo stato 0 si arriva allo stato 1 attraverso il simbolo $P[1]$. Dallo stato $j = m$ si arriva sempre a uno stato $k \leq m$, dallo stato m si può giungere quindi di nuovo allo stato m.

5.2.2 Scansione del testo

La scansione del testo inizia da uno stato iniziale $j_0 = 0$. Partendo da questo stato, leggo il simbolo in posizione i del testo ($T[i]$) e mi sposto nello stato j_i attraverso la funzione di transizione $\delta(j_{i-1}, T[i])$.

Esempio 11. *Consideriamo il testo T:*

1	2	3	4	5	6	7
c	a	b	a	c	a	b

e il pattern P:

1	2	3	4
a	c	a	c

Su cui è stata definita la seguente funzione di transizione δ:

δ	**a**	**b**	**c**
0	1	0	0
1	1	0	2
2	3	0	0
3	1	0	4
4	3	0	0

La scansione del testo T cercando le occorrenze esatte del pattern P mi permette di ottenere il risultato riportato in figura 5.1

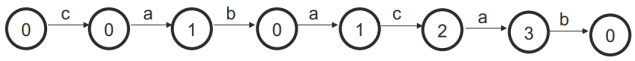

Figura 5.1: Risultato della scansione del testo

A questo punto è necessario identificare un'occorrenza esatta del pattern P nel testo T. Per risolvere questo, partiamo dalla successione di stati che abbiamo definito per la scansione del testo. Essa corrisponde a una successione di posizioni su P, o in altre parole, a una successione di lunghezze di prefissi del pattern P.

Teorema 19. *j_i, con $0 \leq i \leq n$, è la lunghezza del **più lungo** prefisso di P uguale a una sottostringa di T che finisce in posizione i.*

Dimostrazione 14. *È possibile dimostrare il teorema precedente con una dimostrazione per induzione:*

1. ***Caso base**: per lo stato $j_0 = 0$ il teorema è banalmente dimostrabile, in quanto il prefisso di lunghezza 0 è il prefisso nullo che è sottostringa di T.*

2. ***Passo induttivo**: se j_{i-1} è la lunghezza del più lungo prefisso di P uguale alla sottostringa di T che finisce in posizione $i-1$, allora j_i è la lunghezza del più lungo prefisso di P uguale alla sottostringa di T che finisce in posizione i.*

   Per dimostrare il passo induttivo ci basiamo sulla seguente ipotesi: j_{i-1} è la lunghezza del più lungo prefisso di P uguale a una sottostringa di T che finisce in posizione $i-1$.

   - ***Caso 1**: $j_{i-1} < m$ e $P[j_{i-1}+1] = T[i]$ questo implica $j_i = \delta(j_{i-1}, T[i]) = j_{i-1}+1$*
     - *$j_{i-1} \neq 0$: la tesi è confermata*
     - *$j_{i-1} = 0$ vuol dire che il carattere corrisponde con il carattere iniziale del pattern*
   - ***Caso 2**: $j_{i-1} = m$ oppure $P[j_{i-1}+1] \neq T[i]$ questo implica $j_i = \delta(j_{i-1}, T[i]) = k$ dove k è la lunghezza del bordo di $P[1, j_{i-1}]T[i]$.*
     - *$0 < j_{i-1} < m$: in questo caso il valore di k mi rappresenta una parte del testo per cui ho già verificato un'occorrenza esatta, questo mi viene garantito dalla definizione di bordo.*
     - *$j_{i-1} = 0$ vuol dire che sono rimasto nello stato 0.*
     - *$j_{i-1} = m$ ho trovato un'occorrenza esatta del pattern, inoltre per evitare di perdere delle occorrenze mi sposto in base alla lunghezza del bordo del pattern concatenato con il carattere successivo.*

Questo teorema mi fornisce la garanzia che non sto perdendo delle occorrenze. Inoltre, posso trovare la posizione di inizio dell'occorrenza esatta come $i - j + 1$. Nel caso in cui $j_i = m$ ho identificato un'occorrenza esatta del pattern P.

Possiamo riassumere la scansione del testo come:

1. Si parte dallo stato iniziale 0 e si effettua una scansione di T dal primo all'ultimo simbolo.
2. Per ogni posizione i di T si effettua la transizione dallo stato corrente j_c al nuovo stato $j_f = \delta(j_c, T[i])$
3. Ogni volta che lo stato j_f è lo stato accettante (m), viene prodotta in output l'occorrenza $i - m + 1$

Algorithm 3 Algoritmo per la ricerca esatta con Automa a Stati Finiti

```
function ASF_EXACT_OCCURRENCES(δ, T, m)
    n ← |T|
    j ← 0
    for i ← 1 to n do
        j ← δ(j, T[i])
        if j = m then
            Output i − m + 1
        end if
    end for
end function
```

Questo algoritmo viene eseguito in tempo $\theta(n)$.

Esempio 12. *Consideriamo il testo T:*

1	2	3	4	5	6	7	8	9	10	11	12	13
c	a	b	a	c	a	c	b	a	c	a	b	a

e il pattern P:

1	2	3	4	5	6	7
a	c	a	c	b	a	c

Su cui è stata definita la seguente funzione di transizione δ:
Otteniamo la seguente esecuzione dell'algoritmo:

δ	**a**	**b**	**c**
0	1	0	0
1	1	0	2
2	3	0	0
3	1	0	4
4	3	5	0
5	6	0	0
6	1	0	7
7	3	0	0

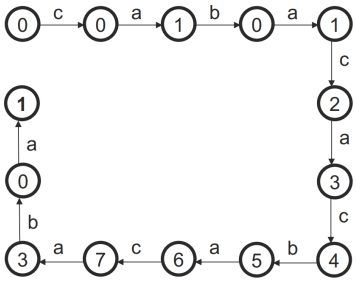

Figura 5.2: Esecuzione dell'algoritmo per la ricerca esatta con Automa a Stati Finiti

5.2.3 Calcolo della funzione di transizione δ

L'algoritmo più semplice che permette di calcolare i valori della funzione di transizione δ consiste nell'applicare la funzione di transizione δ per come è definita senza sfruttare i valori che sono già stati computati. Questo algoritmo richiede un tempo nel caso peggiore pari a $\mathcal{O}(m^3|\Sigma|)$, questo è dovuto al fatto che per calcolare il bordo è necessario un tempo, nel caso peggiore pari a $\mathcal{O}(m^2)$.

Definiamo δ_j come la funzione di transizione di $P[1, j]$, ovvero come una funzione:

$$\delta_j : \{0, 1, \ldots, j\} \times \Sigma \to \{0, 1, \ldots, j\} \tag{5.12}$$

per questa funzione possiamo definire due casi particolari:

- δ_0 ovvero la funzione di transizione di $P[1, 0]$ che per definizione è ε, quindi il valore di tale funzione è sempre 0.
- δ_m ovvero la funzione di transizione di $P[1, m]$ la quale corrisponde precisamente alla funzione di transizione del pattern P.

Possiamo definire il calcolo della funzione di transizione *delta* per un pattern P di lunghezza m utilizzando l'induzione nel seguente modo:

- Caso base: calcolo δ_0

Algorithm 4 Algoritmo banale per il calcolo della funzione di transizione δ

```
function TRIVIAL-BUILD-TRANSITION-FUNCTION(P)
    m ← |P|
    δ ← empty_table(m + 1) × |Σ|
    for j ← 0 to m − 1 do
        δ(j, P[j + 1]) ← j + 1
    end for
    for j ← 0 to m do
        for σ ∈ Σ do
            δ(j, σ) ← |B(P[1, j]σ)|
        end for
    end for
    return δ
end function
```

- Passo induttivo: calcolo δ_j da δ_{j-1} in questo caso dobbiamo distinguere il caso in cui $j = 1$ e i restanti.
  - Nel caso in cui $j = 1$, possiamo definire la funzione di transizione come:
    1. Prendo il valore 0 contenuto nella cella della riga 0 di δ_0 in corrispondenza del simbolo $P[1]$
    2. Sostituisco il valore 0 con il valore 1 (stato successivo a 0).
    3. Aggiungo una nuova riga (corrispondente allo stato 1)
    4. Copio la riga che corrisponde allo stato 0 nella riga che corrisponde allo stato 1
    5. Rinomino δ_0 in δ_1
  - Mentre nel caso in cui $j \neq 1$ possiamo definire la funzione di transizione come:
    1. Prendo il valore k contenuto nella cella della riga $j-1$ di δ_{j-1} in corrispondenza del simbolo $P[j]$
    2. Sostituisco il valore k con il valore j (stato successivo a $j-1$)
    3. Aggiungo una nuova riga (corrispondente allo stato j)
    4. Copio la riga che corrisponde allo stato k nella riga che corrisponde allo stato j
    5. Rinomino δ_{j-1} in δ_j

Nota 9. *Dimostrazione tramite esempi su slide.*

Con queste informazioni possiamo definire un algoritmo che mi permette di calcolare la funzione di transizione δ in tempo $\theta(m \cdot |\Sigma|)$. Tale algoritmo sfrutta le informazioni precedentemente calcolate.

Algorithm 5 Algoritmo per il calcolo della funzione di transizione δ

```
function BUILD-TRANSITION-FUNCTION(P)
    m ← |P|
    δ ← empty_table(m + 1) × |Σ|
    for σ ∈ Σ do
        δ(0, σ) ← 0
    end for
    for j ← 1 to m do
        k ← δ(j − 1, P[j])
        δ(j − 1, P[j]) ← j
        for σ ∈ Σ do
            δ(j, σ) ← δ(k, σ)
        end for
    end for
    return δ
end function
```

5.3 Algoritmo di Knuth-Morris-Pratt

Questo algoritmo per la ricerca esatta è basato su un'analisi del pattern. Si hanno due fasi:

1. **Preprocessing del pattern**: questa fase consiste nel calcolo della prefix function ϕ, che è una funzione che associa ad ogni posizione del pattern la lunghezza del più lungo prefisso del pattern che è anche un suffisso del pattern. Questa funzione è calcolata in tempo lineare rispetto alla lunghezza del pattern $\mathcal{O}(m)$.
2. **Scansione del testo**: questa fase consiste nel confrontare il pattern con il testo cercando di identificare le occorrenze esatte di esso. Questa fase è eseguita in tempo lineare rispetto alla lunghezza del testo $\mathcal{O}(n)$.

La **prefix function** o funzione di fallimento ϕ è definita come segue:

$$\phi : \{0, 1, \dots m\} \to \{-1, 0, \dots m\} \tag{5.13}$$

Essa è definita come segue:

$$\phi(j) = \begin{cases} |B(P[1, j])| & \text{se } 1 \leq j \leq m \\ -1 & \text{se } j = 0 \end{cases} \tag{5.14}$$

Esempio 13. *Calcoliamo ϕ sul pattern $P = abcabaabcabab$ con $m = 13$.*

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
ϕ	-1	0	0	0	1	2	1	1	2	3	4	5	6	2

Questo algoritmo è un'evoluzione dell'algoritmo banale per la ricerca delle occorrenze di un pattern in un testo. In particolare, l'algoritmo KMP consiste in:

1. Viene usata una finestra W di lunghezza m che scorre sul testo T da sinistra a destra con posizione iniziale $i = 1$.
2. Si confrontano i simboli di P con i corrispondenti simboli di T all'interno della finestra W andando da sinistra a destra e partendo dal primo simbolo di P.
3. Non appena si incontra un mismatch oppure ogni simbolo di P ha un match con il corrispondente simbolo in W (i è occorrenza esatta), W viene spostata a destra nella posizione $p = i + j - \phi(j - 1) - 1$, dove j è l'indice del simbolo di P che ha causato il mismatch, mentre i è la posizione iniziale di W.
4. L'ultima posizione di W è $n - m + 1$.

Riassumendo:

- W viene spostata dalla posizione i alla posizione p, dove:

  $$p = i + j - \phi(j - 1) - 1$$

  con j indice del simbolo di P che ha causato il mismatch per W in posizione i.
- Il confronto riparte dal simbolo di P in posizione $j = \phi(j - 1) + 1$ e dal simbolo di T in posizione $i + j - 1$.

Nel caso in cui $j = 1$, allora $p = i + 1$ e il confronto riparte dal primo simbolo di P e dal simbolo di T in posizione $i + 1$.

Nota 10. *Chiaramente il confronto riparte dalle posizioni $i+1$ su T e 1 su P, ma dire che riparte dalla posizione i su T e dalla posizione 0 su P implicitamente fa riferimento ad un confronto iniziale fittizio tra $T[i]$ e $P[0]$ (simbolo inesistente) che di default viene considerato un match.*

Nel caso in cui $j = m + 1$, allora $p = i + m - \phi(m)$ e il confronto riparte dalla posizione $\phi(m) + 1$ di P e dal simbolo di T in posizione $i + m$. Le differenze tra l'algoritmo basato sull'automa e KMP sono:

- Automa:
  - Efficiente per pattern piccoli, perché la memoria è contenuta quindi le costanti all'interno del calcolo del tempo sono migliori.
  - Richiede più tempo e memoria per pattern grandi, perché sprechiamo tempo nel calcolo del δ.
  - Ricerca di P in testi diversi utilizzando lo stesso automa

  KMP:
  - Efficiente per pattern grandi
  - Richiede più tempo per pattern piccoli

	Automa a stati finiti	KMP
Preprocessing di P	$\mathcal{O}(m\|\Sigma\|)$	$\mathcal{O}(m)$
Scansione di T	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Spazio	$\mathcal{O}(m\|\Sigma\|)$	$\mathcal{O}(m)$

5.4 Algoritmo di Baeza-Yates e Gonnet

La ricerca esatta effettuata attraverso questo algoritmo effettua un confronto tra i simboli del pattern e del testo in maniera non esplicita, ovvero non confronta carattere per carattere. In questo algoritmo vengono effettuate in parallelo operazioni bit a bit su word di bit, viene anche chiamato *algoritmo bit parallel*.

Questo algoritmo segue il paradigma **shift-and** ovvero compie fondamentalmente due sole operazioni:

- **Shift** dei bit.
- **AND** logico tra i bit.

Come gli algoritmi visti fin ora, anche in questo caso possiamo descrivere il suo funzionamento attraverso due fasi:

- **Preprocessing del pattern** P nel quale vengono calcolate $|\Sigma|$ **words** ognuna di m bit. Questa operazione viene eseguita in tempo $\theta(|\Sigma| + m)$.
- **Scansione del testo** T per cercare le occorrenze esatte del pattern P. Questa operazione viene eseguita in tempo $\theta(n)$.

5.4.1 Word e Operatori

Definizione 36 (**Word di bit**). *Una **word di bit** è un gruppo di bit che viene trattato come un'unità la cui dimensione può variare e che rappresenta un valore di un certo tipo, come ad esempio un numero o un carattere. In una word, il bit più a destra è quello meno significativo, mentre quello più a sinistra è quello più significativo.*

Sulle word si possono eseguire delle operazioni *bit a bit*, ovvero si esegue un'operazione tra i bit corrispondenti di due o più words di bit della stessa lunghezza. Il valore restituito da queste operazioni è una nuova word in cui ogni bit è il risultato dell'operazione tra i bit corrispondenti nelle word in input. Tra queste operazioni abbiamo:

- **Congiunzione logica** → AND. Questa operazione è implementata come:

$$w = w_1 \text{ AND } w_2 \tag{5.15}$$

  restituisce una word w tale che:
  - $w[j] = 1$ se e solo se $w_1[j]$ AND $w_2[j] = 1$.
  - $w[j] = 0$ altrimenti.

- **Disgiunzione logica** (inclusiva) → OR. Questa operazione è implementata come:

$$w = w_1 \text{ OR } w_2 \tag{5.16}$$

  restituisce una word w tale che:
  - $w[j] = 1$ se e solo se $w_1[j]$ OR $w_2[j] = 1$.
  - $w[j] = 0$ altrimenti.

- **Shift dei bit** di una posizione a destra con bit più significativo a 0 → RSHIFT. Questa operazione è implementata come:

$$w = \text{RSHIFT}(w_1) \tag{5.17}$$

  restituisce una word w tale che:
  - $w[j] = w_1[j-1]$ se $j \geq 2$.
  - $w[1] = 0$ altrimenti.

- **Shift dei bit** di una posizione a destra con bit più significativo a 1 → RSHIFT1. Questa operazione viene implementata come un RSHIFT seguito da un OR con una maschera in cui nella prima posizione è presente 1 e nelle altre 0.

5.4.2 Preprocessing del pattern

Dato un pattern di lunghezza m e σ in Σ, B_σ è una word di m bit tale che:

$$B_\sigma[j] = 1 \iff P[j] = \sigma \tag{5.18}$$

Viene creata una word per ogni simbolo dell'alfabeto Σ e viene memorizzata in una tabella B. Con questa rappresentazione posso effettuare le query del tipo: "il simbolo in posizione j di P è uguale a un certo simbolo σ".

Esempio 14. *Calcoliamo la tabella B per un pattern $P = abcaba$ su un alfabeto $\Sigma = \{a, b, c, d\}$.*

P	a	b	c	a	b	a
B_a	1	0	0	1	0	1
B_b	0	1	0	0	1	0
B_c	0	0	1	0	0	0
B_d	0	0	0	0	0	0

Vediamo ora come calcolare la tabella B:

1. tutte le word B_σ vengono inizializzate a m bit a 0.
2. viene creata una maschera M di m bit tutti uguali a 0 tranne il più significativo che è uguale a 1.
3. si esegue una scansione di P da sinistra a destra, e per ogni posizione j vengono eseguite le due operazioni bit a bit:
   - $B_{P[j]} = M$ **OR** $B_{P[j]}$
   - $M =$ **RSHIFT**(M)

L'algoritmo per calcolare la tabella richiede un tempo pari a $\theta(|\Sigma| + m)$ ed è il seguente:

Algorithm 6 Algoritmo per il calcolo della tabella B

```
function COMPUTE-TABLE-B(P)
    m ← |P|
    B ← empty table of |Σ| words B_σ
    for σ ∈ Σ do
        B_σ ← 00...0
    end for
    M ← 10...0
    for σ ∈ Σ do
        σ ← P[j]
        B_σ ← M OR B_σ
        M = RSHIFT(M)
    end for
    return B
end function
```

5.4.3 Scansione del testo

La procedura per la scansione del testo è rappresentata come:

1. Il testo T viene scandito dalla prima all'ultima posizione.
2. Per ogni posizione i del testo T viene calcolata una word D_i di m bit.
3. Ogni volta che in D_i il bit meno significativo è uguale a 1, allora $i - m + 1$ è occorrenza esatta di P in T.

Dobbiamo ora definire cosa si intende con la word D_i. Prima di fare ciò dobbiamo definire $P[1, j] = suff(T[1, i])$ ovvero $P[1, j]$ è uguale a un suffisso di $T[1, i]$.

Definizione 37 (**Word** D_i)**.** *Dati P lungo m e T lungo n, D_i $(0 \leq i \leq n)$ è una word di m bit tale che:*

$$D_i[j] = 1 \iff P[1, j] = suff(T[1, i]) \tag{5.19}$$

Inoltre, sappiamo per definizione che:

- $D_0 = 00 \dots 0$ *dato che* $P[1, j] \neq suff(T[1, 0])\ \forall j$
- $D_i[m] = 1$ *se e solo se* $P[1, m] = suff(T[1, i]) = T[i - m + 1, i]$ *ovvero si ha un'occorrenza esatta di P in T nella posizione $i - m + 1$.*

Nota 11. *Nota che da D_i possiamo anche ottenere la lunghezza del bordo del pattern quando LSB di D_i è a 1 allora la lunghezza del bordo coincide con la posizione del bit a 1 più più vicino all'LSB.*

Ovviamente calcolare D_i è sempre troppo costoso, infatti, D_i viene sempre ottenuto a partire da D_{i-1}, portando a modificare l'algoritmo di scansione del testo nel seguente modo:

- Inizia dalla word $D_0 = 00 \dots 0$
- Per ogni i da 1 a n calcola la word D_i a partire dalla word D_{i-1}.
- Ogni volta che D_i ha il bit meno significativo uguale a 1, viene prodotta in output l'occorrenza $i - m + 1$.

Vediamo ora come si calcola il valore di D_i a partire da D_{i-1}:

- Se $j = 1$ allora posso calcolarla come:

  $$D_i[1] = 1 \mathbf{AND} B_{T[i]}[1] \tag{5.20}$$

  Perché $D_i[1] = 1 \iff P[1, 1] = suff(T[1, i]) \iff P[1] = T[i] \iff B_{T[i]}[1]$. Si aggiunge 1 per semplificare le operazioni bit a bit.

- Se $j > 1$ allora posso calcolarla come:

  $$D_i[j] = D_{i-1}[j - 1] \mathbf{AND} B_{T[i]}[j] \tag{5.21}$$

  Perché $D_i[j] = 1 \iff P[1, j] = suff(T[1, i]) \iff P[1, j - 1] = suff(T[1, i - 1]) \mathbf{AND} P[j] = T[i] \iff D_{i-1}[j - 1] \mathbf{AND} B_{T[i]}[j]$.

In questo modo stiamo ancora aggiornando un bit alla volta, ma sfruttando le operazioni bit a bit si possono aggiornare tutti in tempo **costante** nel seguente modo:

$$D_i = \mathbf{RSHIFT1}(D_{i-1})\ \mathbf{AND}\ B_{T[i]} \tag{5.22}$$

Vediamo ora come implementare l'algoritmo che effettua la scansione del testo in tempo $\theta(n)$

Algorithm 7 Algoritmo per la scansione del testo

```
function BYG(B, T)
    n ← |T|
    D ← 00...0
    M ← 00...01
    for i ← 1 to n do
        σ ← T[i]
        D ← RSHIFT1(D_{i-1}) AND B_{T[i]}
        if (D ANDM) = M then
            output i − m + 1
        end if
    end for
end function
```

Esempio 15. *Siano un testo $T = babcabaadc$ e un pattern $P = abcaba$ tale che $|T| = 10$ e $|P| = 6$. Entrambe le stringhe costruite su un alfabeto $\Sigma = \{a, b, c, d\}$, allora costruiamo la tabella B_σ: I singoli passi sono i seguenti:*

P	a	b	c	a	b	a
B_a	1	0	0	1	0	1
B_b	0	1	0	0	1	0
B_c	0	0	1	0	0	0
B_d	0	0	0	0	0	0

- *Quindi partiremo da* $D_0 = 000000$*, leggo il primo simbolo del testo* $T[1] = b$ *e poi calcolo*

$$D_1 = RSHIFT1(D_0)\ AND\ B_b = 100000\ AND\ 010010 = 000000$$

- *leggo il secondo simbolo del testo* $T[2] = a$ *e poi calcolo* D_2

$$D_2 = 100000\ AND\ 100101 = 100000$$

- *leggo il secondo simbolo del testo* $T[3] = b$ *e poi calcolo* D_3

$$D_3 = 110000\ AND\ 010010 = 010000$$

- *leggo il secondo simbolo del testo* $T[4] = c$ *e poi calcolo* D_4

$$D_4 = 101100\ AND\ 001000 = 001000$$

- *leggo il secondo simbolo del testo* $T[5] = a$ *e poi calcolo* D_5

$$D_5 = 100100\ AND\ 100101 = 100100$$

- ...
- *leggo il secondo simbolo del testo* $T[7] = a$ *e poi calcolo* D_7

$$D_6 = 101001\ AND\ 100101 = 100001$$

- *Ho il bit LSB di* D_6 *uguale a* 1*, quindi ho trovato un match che corrisponde a* $i - m + 1 = 7 - 6 + 1 = 2$*. Poi continuo fino a quando non termino* T*.*

5.5 Algoritmo di Wu e Manber

L'algoritmo risolvere problemi di **stringmatching approssimati** e il funzionamento è simile a **BYG**. Riprendiamo la definizione di ricerca approssimata di un pattern P in un testo T:

Definizione 38 (**Occorrenza approssimata**)**.** *Una posizione* i *del testo* T *tale che esista una sottostringa* $S = T[i - L + 1]$ *tale che* $ED(P, S) \leq k$ *è detta* ***occurrenza approssimata*** *di* P *in* T*.*

Come gli algoritmi visti fino a questo momento, quello di Wu e Manber è basato su due fasi:

1. **prepocessing** del pattern P nel quale vengono calcolate $|\Sigma|$ **words** ognuna di m bit. Questa operazione viene eseguita in tempo $\theta(|\Sigma| + m)$. (*uguale a quello dell'algoritmo BYG*)
2. **Scansione del testo** T per cercare le occorrenze approssimate del pattern P. Questa operazione viene eseguita in tempo $\theta(k \cdot n)$.

5.5.1 Scansione del testo

Dato che la fase di preprocessing è equivalente a quella per l'algoritmo BYG, passiamo subito alla scansione del testo.

Definizione 39. *Definiamo* $P[1, j] = suff_h(T[1, i])$ *ovvero* $P[1, j]$ *è uguale a un suffisso di* $T[1, i]$ *a meno di* h *errori. In altre parole, riesco a trovare un suffisso* S *di* $T[1, i]$ *tale che* $ED(P[1, j], S) \leq h$*.*

Estenderemo la definizione di D_i di ShiftAND al caso in cui si ammettono al più h errori.

Definizione 40 (**Word** D_i^h)**.** *Dati un pattern P lungo m, un testo T lungo n e una soglia di errore k, possiamo definire la* ***word*** D_i^h *come una word di m bit tale che:*

$$D_i^h[j] = 1 \iff P[1,j] = suff_h(T[1,i]) \tag{5.23}$$

Vediamo ora alcuni casi particolare:

- D_i^0: ovvero la la word con $h = 0$, è per definizione la word D_i dell'algoritmo BYG.
- D_0^h: ovvero la word con $i = 0$, è tale che:
  - $j \leq h \implies D_0^h[j] = 1$, questo perché $P[1,j]$ è un prefisso di $T[1,0]$ e quindi $ED(P[1,j], T[1,0]) \leq h$.
  - $j > h \implies D_0^h[j] = 0$, questo perché $P[1,j]$ non è un prefisso di $T[1,0]$ e quindi $ED(P[1,j], T[1,0]) > h$.

  Si ottiene quindi una word dove i primi h bit sono uguali a 1 e i restanti $m - h$ bit sono posti a 0.
- $D_i^h[m] = 1 \iff P[1,m] = suff_h(T[1,i])$, ovvero se $P[1,m]$ è un suffisso di $T[1,i]$ a meno di h errori, allora $D_i^h[m] = 1$. Nel caso particolare in cui $h = 0$ si ha un occorrenza esatta in $i - m + 1$. Mentre nel caso in cui $h = k$ si ha un'occorrenza approssimata in posizione i.

Nota 12. *In generale, $D_i^{h'}[j] = 1$ implica $D_i^h[j] = 1$ con $h > h'$. Inoltre, $D_i^h[j] = 1$ implica $D_i^{h+1}[j+1] = 1$ e $D_i^{h+1}[j-1] = 1$*

Definita la word D_i^h, possiamo definire la scansione del testo T come una successione di $k+1$ scansioni, dove solamente all'iterazione k si vanno a verificare i bit finali delle word per cercare l'occorrenza approssimata. Nello specifico, se all'iterazione k si ha che $D_i^k[m] = 1$, allora si ha in output l'occorrenza approssimata i.

Fino ad ora abbiamo definito come calcolare il caso in cui $h = 0$ oppure il caso in cui $i = 0$. Vediamo ora come calcolare il caso in cui $i > 0, h > 0$.

Partiamo considerando il caso in cui $j > 1$, in questa situazione abbiamo che:

$$D_i^h[j] = 1 \iff P[1,j] = suff_h(T[1,i]) \Leftarrow P[1,j-1] = suff_h(T[1,i-1]) \textbf{ AND } T[i] = P[j] \tag{5.24}$$

ovvero $D_i^h[j] = 1$ se e solo se $P[1,j-1]$ è un suffisso di $T[1,i-1]$ a meno di h errori e $T[i] = P[j]$. Questo significa che $D_i^h[j] = 1$ se e solo se:

$$D_{i-1}^h[j-1] = D_{i-1}^h[j-1] = 1 \textbf{ AND } B_{T[i]}[j] = 1 \tag{5.25}$$

Quello appena visto non è l'unico caso, infatti possiamo avere anche:

$$D_i^h[j] = 1 \iff P[1,j] = suff_h(T[1,i]) \Leftarrow P[1,j-1] = suff_{h-1}(T[1,i-1]) \tag{5.26}$$

ovvero non ho ancora raggiunto il limite di errori e quindi posso ancora avere errori senza superare la soglia h. Questo significa che $D_i^h[j] = 1$ se e solo se:

$$D_i^h[j] = 1 \Leftarrow D_{i-1}^{h-1}[j-1] = 1 \tag{5.27}$$

Un altra casistica è data dal fatto che posso avere un carattere in meno nel testo, ottenendo quindi:

$$D_i^h[j] = 1 \iff P[1,j] = suff_h(T[1,i]) \Leftarrow P[1,j] = suff_{h-1}(T[1,i-1]) \tag{5.28}$$

ovvero non ho ancora raggiunto il limite di errori. Questo significa che $D_i^h[j] = 1$ se e solo se:

$$D_i^h[j] = 1 \Leftarrow D_{i-1}^{h-1}[j] = 1 \tag{5.29}$$

Il quarto e ultimo caso è dato dal fatto che posso modificare il pattern, ovvero:

$$D_i^h[j] = 1 \iff P[1,j] = suff_h(T[1,i]) \Leftarrow P[1,j-1] = suff_{h-1}(T[1,i]) \tag{5.30}$$

ovvero non ho ancora raggiunto il limite di errori. Questo significa che $D_i^h[j] = 1$ se e solo se:

$$D_i^h[j] = 1 \Leftarrow D_i^{h-1}[j-1] = 1 \tag{5.31}$$

A questo punto posso riassumere queste casistiche in un'unica formula:

$$D_i^h[j] = (D_{i-1}^h[j-1] \textbf{ AND } B_{T[i]}[j]) \textbf{ OR } D_{i-1}^{h-1}[j-1] \textbf{ OR } D_{i-1}^{h-1}[j] \textbf{ OR } D_i^{h-1}[j-1] \tag{5.32}$$

Manca ora da analizzare il caso in cui $j = 1$, in questo caso se sostituiamo $j = 1$ alla formula appena trovata otteniamo dei valori che non possiamo conoscere (come ad esempio $D^h_{i-1}[0]$). Quindi, per risolvere questo problema, sostituiamo questi valori con 1, ottenendo:

$$D^h_i[j] = (1 \textbf{ AND } B_{T[i]}[j]) \textbf{ OR } 1 \textbf{ OR } D^{h-1}_{i-1}[j] \textbf{ OR } 1 \tag{5.33}$$

Definiendo la formula in questo modo possiamo sostituire tutte le operazioni che controllano il bit in posizione $j-1$ con l'operazione RSHIFT1, ottenendo:

$$D^h_i[j] = (RSHIFT1(D^h_{i-1}[j]) \textbf{ AND } B_{T[i]}[j]) \textbf{ OR } RSHIFT1(D^{h-1}_{i-1}[j]) \textbf{ OR } D^{h-1}_{i-1}[j] \textbf{ OR } RSHIFT1(D^{h-1}_i[j]) \tag{5.34}$$

Vediamo ora come è possibile implementare l'algoritmo per la scansione del testo:

1. Inizializzazione della parola D^0_0 a $00\ldots0$.
2. Calcolo di D^0_i per $i = 1, \ldots, n$, utilizzando l'algoritmo di BYG.
3. Per h da 1 a k:
   (a) Calcolo di D^h_0.
   (b) Per i da 1 a n:
      i. Calcolo di D^h_i a partire da D^h_{i-1}.
      ii. Se $D^h_i[m] = 1$ **AND** $h = k$ allora output i.

5.6 Strutture di indicizzazione del testo

Per rendere più efficiente la ricerca di un pattern esatto si effettuano delle operazioni di preprocessing del testo che permettono di ristrutturarlo per accedere velocemente alle sue porzioni.

Le strutture di indicizzazione sono strutture che manipolano l'input in modo da rendere più efficiente le operazioni su di esso, ma hanno il problema che richiedono molta memoria. Inoltre, la loro costruzione risulta complessa.

Queste strutture si basano su:

- **Ordine lessicografico**: si definisce un ordine lessicografico dei caratteri dell'alfabeto per poter determinare un ordine delle parole.
- **Carattere terminatore** \$: carattere più piccolo dell'alfabeto che serve come carattere per identificare il termine della stringa.

In aggiunta dovremo definire:

Definizione 41 (**q-suffisso**). *Il suffisso di indice q (q-suffisso) è il suffisso che inizia nella posizione q del testo, quindi* $T[q:] = T[q : |T|]$

Il suffisso del carattere terminatore è il carattere terminatore stesso.

Definizione 42 (**Ordinamento dei suffissi**). *Definiremo l'ordinamento dei suffissi nel segunete modo. Dati due suffissi* s *e* s'*, sia* i *la più piccola posizione tale che* $s[i] \neq s'[i]$*, dove:*

- $s \prec s' \iff s[i] < s'[i]$
- $s \succ s' \iff s[i] > s'[i]$

5.6.1 Suffix Array

Questa struttura di indicizzazione è stata inventata da Myers e Manber nel 1990, non contiene l'informazione sui simboli del testo e occupa $\Theta(n \log n)$ spazio. Si può effettuare la ricerca esatta di un pattern P su un testo T in $\mathcal{O}(|P| \log |T|)$.

Definizione 43 (**Suffix Array**). *Il* ***suffix array*** *di un testo* T *lungo* n *è un array* S *lungo* n*, tale che* $S[i] = q$ *se e solo se il q-suffisso è l'i-esimo suffisso nell'ordinamento lessicografico dei suffissi di* T*.*

SA	1	2	3	4	5	6	7	8	9
1	g	g	t	c	a	g	t	c	$
2	g	t	c	a	g	t	c	$	
3	t	c	a	g	t	c	$		
4	c	a	g	t	c	$			
5	a	g	t	c	$				
6	g	t	c	$					
7	t	c	$						
8	c	$							
9	$								

Esempio 16. *Consideriamo il testo $T = ggtcagtc\$$, il cui ordinamento lessicografico è definito come:*

$$\$ < a < c < g < t \tag{5.35}$$

Possiamo costruire il suffix array S come:

- *Generiamo tutti i suffissi di T:*
- *Ordiniamo i suffissi in ordine lessicografico:*

SA	1	2	3	4	5	6	7	8	9
9	$								
5	a	g	t	c	$				
8	c	$							
4	c	a	g	t	c	$			
1	g	g	t	c	a	g	t	c	$
6	g	t	c	$					
2	g	t	c	a	g	t	c	$	
7	t	c	$						
3	t	c	a	g	t	c	$		

- *Otteniamo il suffix array S come: $S = [9, 5, 8, 4, 1, 6, 2, 7, 3]$*

Possiamo interpretare il suffix array come un array di indici, dove ogni elemento $S[i]$ rappresenta la posizione del suffisso i-esimo nell'ordinamento.

Lo spazio richiesto per la memorizzazione del suffix array è $\Theta(n \log n)$.

Ricerca esatta

Vediamo ora il funzionamento della ricerca esatta di un pattern P di lunghezza m in testo T di lunghezza n.

La ricerca esatta di un pattern P in un testo T avviene in due fasi:

1. Preprocessing del testo T per costruire il suffix array S.
2. Ricerca del pattern P nel testo T utilizzando il suffix array S. L'operazione di ricerca avviene in tempo $\theta(m \log n)$.

Per prima cosa si può osservare che:

- Se P occorre k volte in T allora P è prefisso di k suffissi di T. Ad esempio, dato il testo $T = xabxab\$$ e il pattern $P = ab$ possiamo osservare che P occorre 2 volte in T il che implica che P è prefisso di due suffissi di T ovvero 2-suffisso e 5-suffisso.
- Gli indici dei k suffissi sono le occorrenze di P e sono consecutivi nel suffix array. Ad esempio, dato il testo $T = xabxab\$$ possiamo calcolare il suo suffix array e ottenere $S = [7, 5, 2, 6, 3, 4, 1]$. Se consideriamo il pattern $P = ab$ possiamo osservare che esso occorre 2 volte in T nelle posizioni 2 e 5 che sono consecutive.
- Se P è prefisso del q-suffisso ed è minore (maggiore) di un q'-suffisso, allora anche il q-suffisso sarà minore (maggiore) del q'-suffisso.

Quindi l'algortmo di ricerca sarà:

1. Si inizializza un intervallo di posizioni $[L, R] = [1, |T|]$
2. Si considera il suffisso di indice $S[p]$ con $p = \lfloor \frac{R-L}{2} \rfloor$,in tempo $O(m)$ si controlla:
   - $P \prec S[p]$-suffisso: si ripete il punto 2 ponendo $[L, R] = [L, p]$
   - $P \succ S[p]$-suffisso: si ripete il punto 2 ponendo $[L, R] = [p + 1, R]$
   - P occorre come prefisso in $S[p]$-suffisso: si termina e si restituisce la posizione $S[p]$

Questo algoritmo trova la prima occorrenza del P in tempo $O(|P| \log |T|)$.

Algorithm 8 Algoritmo di ricerca esatta di un pattern

```
function SuffixArraySearch(T, P)
    n ← |T|
    m ← |P|
    S ← SuffixArray(T)
    l ← 1
    r ← n
    while l ≤ r do
        p ← ⌊(l + r)/2⌋
        if P < S[p]−suffisso then
            r ← p − 1
        else if P > S[p]−suffisso then
            l ← p + 1
        else
            return p
        end if
    end while
    return −1
end function
```

5.6.2 Burrows Wheeler Transform

Proposta da Burrows e Wheeler nel '94, costruisce una permutazione reversibile dei simboli del testo, occupa uno spazio di $\Theta(|T|log|\Sigma|)$ ed è usata in B-zip 2.

Prima di definire la trasformata di Burrows-Wheeler, definiamo il concetto di **rotazione di indice** q.

Definizione 44 (**q-rotazione**). *La rotazione di indice q (q-rotazione) è la concatenazione del q- suffisso* $T[q, |T|]$ *e del prefisso* $T[1, q - 1]$.

Esempio 17. *Dato il seguente testo* $T = ggtcagtc\$$*, allora la* 3*-rotazione è:*

$$tcagtc\$gg$$

Mentre la 9*-rotazione coincide con:*

$$\$ggtcagtc$$

Mentre la 1*-rotazione coincide con:*

$$ggtcagtc\$$$

Possiamo osservare che la q-rotazione r è una stringa lunga n che ha:

1. il q-suffisso come prefisso.
2. il prefisso lungo $q - 1$ come suffisso.

Questo significa che:

- $r[1]$ è uguale a $T[q]$ (primo simborlo del q-suffisso)
- $r[n]$ è uguale a $T[q - 1]$ se $q > 1$, questo implica che:

  - L'ultimo simbolo della q-rotazione è il simbolo iniziale della $q-1$-rotazione
  - il $q-1$-suffisso contiene il q-suffisso come suffisso se $q > 1$
- $r[n]$ è uguale a $T[n]$ se $q = 1$, questo implica che:
  - l'ultimo simbolo della q-rotazione è il simbolo iniziale della n-rotazione (e quindi dell'n-suffisso)

Definiremo anche una regola di ordinamento

Definizione 45 (**Ordinamento delle rotazioni**). *Date due rotazioni r e r', sia i la più piccola posizione tale che $r[i] \neq r'[i]$, dove:*

- $r \prec r' \iff r[i] < r'[i]$
- $r \succ r' \iff r[i] > r'[i]$

Possiamo introdurre questi teoremi

Teorema 20. *Sia q-rotazione r_1 una p-rotazione r_2 tale che:*

- $r_1 \prec r_2$
- $r_1[n] = r_2[n]$

Allora $(q-1)$-rotazione r'_1 è minore della $(p-1)$-rotazione r'_2

Teorema 21. *Sia q-rotazione r_1 una p-rotazione r_2 tale che $r_1[n] \prec r_2[n]$, Allora $(q-1)$-rotazione r'_1 è minore della $(p-1)$-rotazione r'_2*

Dopo questo preambolo si può definire

Definizione 46 (**Burrows Wheeler Transform**). *La BWT B di un testo T lungo n è un array di lunghezza n tale che $B[i] = r_i[n]$ se e solo se r_i è l'i-esima rotazione nell'ordinamento lessicografico delle rotazioni di T.*

In altri termini, $B[i]$ è l'ultimo simbolo della rotazione r_i che è l'i-esima rotazione nell'ordinamento lessicografico e che sarà la q-rotazione per un certo valore q.

Ecco un esempio di costruzione:

Esempio 18. *Sia $T = ggtcagtc\$$. Elenchiamo tutte le rotazioni del testo*

BWT	1	2	3	4	5	6	7	8	9
1	g	g	t	c	a	g	t	c	$
2	g	t	c	a	g	t	c	$	g
3	t	c	a	g	t	c	$	g	g
4	c	a	g	t	c	$	g	g	t
5	a	g	t	c	$	g	g	t	c
6	g	t	c	$	g	g	t	c	a
7	t	c	$	g	g	t	c	a	g
8	c	$	g	g	t	c	a	g	t
9	$	g	g	t	c	a	g	t	c

ordiniamo le rotazioni secono l'ordine lessico grafico:
Quindi la BWT è data dall'ultima colonna della tabella:

$$B = [c, c, t, t, \$, a, g, g, g] \tag{5.36}$$

Lo spazio richiesto per la memorizzazione della trasformata di Burrows-Wheeler è $\theta(n \log |\Sigma|)$.

Definizione 47. *Definiamo F come l'array di lunghezza n dei simboli iniziali delle rotazioni ordinate. Tale array fornisce sempre l'ordinameto lessicografico dei simboli del testo.*

Sia $B[i] = r_i[n]$ con r_i i-esima più piccola rotazione nell'ordinamento lessicografico. Se r_i è la q-rotazione di T, allora $B[i]$ è l'ultimo simbolo della q-rotazione, cioè $T[q-1]$.

Il primo simbolo della q-rotazione è il simbolo $T[q]$ e coincide con il simbolo in posizione $F[i]$ del vettore F.

Della BWT possiamo riconoscere le seguenti proprietà:

BWT	1	2	3	4	5	6	7	8	9
9	\$	g	g	t	c	a	g	t	c
5	a	g	t	c	\$	g	g	t	c
8	c	\$	g	g	t	c	a	g	t
4	c	a	g	t	c	\$	g	g	t
1	g	g	t	c	a	g	t	c	\$
6	g	t	c	\$	g	g	t	c	a
2	g	t	c	a	g	t	c	\$	g
7	t	c	\$	g	g	t	c	a	g
3	t	c	a	g	t	c	\$	g	g

- **Proprietà 1**: per ogni posizione i, il simbolo $B[i]$ precede nel testo il simbolo $F[i]$. Se $B[i] = \$$, allora $F[i] = T[1]$, inoltre, $B[1] \neq \$ = T[n-1]$ e $F[1] = \$$
- **Proprietà 2** (**Last-First mapping**): l'r-esimo simbolo σ in B e l'r-esimo simbolo simobolo σ in F sono lo stesso simbolo del testo T.

Nota 13. *Il primo simbolo della BWT è l'ultimo del testo prima di* \$,

Esempio 19.

$$B = [c, c, t, t, \$, a, g, g, g] \tag{5.37}$$

Allora $B[4]$ è la 4-rotazione che termina con il carattere $B[4]$ il quale coincide con il carattere in posizione $F[9]$.

La proprietà 2 viene implementata con la **Last-First function**.

Definizione 48 (**Last-First function**)**.** *La **Last-First function** è una funzione definita come:*

$$j = LF(i) \tag{5.38}$$

tale che il simbolo $B[i]$ e il simbolo $F[j]$ sono lo stesso simbolo.

Questa funzione si deve calcolare in tempo costante.
Utilizzando le due proprietà possiamo ricostruire il testo T partendo dalla BWT e dal vettore F.

Esempio 20 (Ricostruzione del testo T usando BWT e le proprietà. (Esercizio di esame))**.** *Per la ricostruzione del testo T procede costruendolo dal fondo. Si inizia ricavando F da B, questo avviene ordinando, secondo l'ordinamento lessicografico, il vettore BWT.*
Ottenuto il vettore F si procede a ricostruire il testo T partendo dal fondo nel seguente modo:

- *Si parte dal simbolo* \$*, che è il primo elemento nell'array F.*
- *Da questo utilizzando la proprietà 1 si trova il simbolo $B[1]$ che precede* \$*.*
- *A questo punto, utilizzando la proprietà 2 si trova la posizione del carattere $B[1]$ in F, ovvero $LF(1)$.*
- *Si ripete il procedimento fino a quando non si arriva al simbolo* \$*.*

Riassumendo:

- Il simbolo $B[i]$ della BWT è l'ultimo simbolo della i-esima rotazione r_i.
- Supponiamo che r_i sia la rotazione che inizia in posizione q del testo, cioè q-rotazione.
- L'ultimo simbolo di r_i è il simbolo $T[q-1]$.
- Quindi $B[i]$, che è l'ultimo simbolo di r_i, sarà il simbolo $T[q-1]$.
- La rotazione r_i inizia con il q-suffiso e quindi $B[i]$ è il simbolo che precede q-suffiso.
- Sicuramente, il q-suffisso e l'i-esimo nell'ordinamento lessicografico dei suffissi di T (per il teorema introdotto), si ha questa sincronizzazione per il \$.
- In conclusione, $B[i]$ è il simbolo che precede l'i-esimo suffisso nell'ordinamento lessicografico dei simboli di T.

Attraverso queste osservazioni, possiamo affermare che BWT e SA sono in relazione, più precisamente si ha una corrispondenza degli indici delle r_i, vero per il simbolo \$. Quindi

$$B[i] = T[S[i] - 1]\ T[S[i]] = F[i] \tag{5.39}$$

Last-First mapping può essere definita come il suffisso che inizia con l'r-esimo simbolo σ della BWT è l'r-esimo suffisso di T che inizia con il simbolo σ nell'ordinamento lessicografico dei suffissi di T.

La funzione $j = LF(i)$ fornisce la posizione j del suffisso che inizia con $B[i]$.

Possiamo quindi definire la procedeura di calcolo della BWT partendo dal suffix array come:

Algorithm 9 Algoritmo per il passaggio da SA a BWT

```
function BWT(T)
    n ← |T|
    S ← SuffixArray(T)
    B ← ∅
    for i ← 1 to n do
        B[i] ← T[S[i] − 1]
    end for
    return B
end function
```

Ricerca esatta con BWT

Sia Q una stringa definita su un alfabeto Σ ($Q \in \Sigma^*$) possiamo definire il Q-intervallo.

Definizione 49 (**Q-intervallo rispetto alla BWT**). *Q-intervallo è l'intervallo $[b, e)$ di posizioni della BWT che contengono i simboli che precedono i suffissi che condividono la striga Q come prefisso.*

Definizione 50 (**Q-intervallo rispetto al SA**). *Q-intervallo è l'intervallo $[b, e)$ di posizioni del SA che contengono gli indici dei suffissi che condividono la striga Q come prefisso*

Si può osservare che per $Q = \varepsilon$ si ha il ε-intervallo, il quale coicide con l'intervallo $[1, n + 1)$.

Esempio 21. *Dato l'alfabeto ordinato $\Sigma = \{\$, a, c, g, t\}$ e il testo $T = acaaacatat\$$ allora possiamo costruire il SA e BWT.*

Indici	SA	suffissi ordinati	BWT
1	11	\$	t
2	3	aaacatat\$	c
3	4	aacatat\$	a
4	1	acaaacatat\$	\$
5	5	acatat\$	a
6	9	at\$	t
7	7	atat\$	c
8	2	caaacatat\$	a
9	6	catat\$	a
10	10	t\$	a
11	8	tat\$	a

possiamo effettuare le seguenti query:

- *aca-intervallo equivale a* $[4, 6)$
- *a-intervallo equivale a* $[2, 8)$
- *cat-intervallo equivale a* $[9, 10)$
- *ε-intervallo equivale a* $[1, 12)$

Dato un Q-intervallo $[b, e)$ si possono effettuare le seguenti osservazioni:

- Dato un suffix array S,possiamo trovare gli indici dei suffissi che condividono Q come prefisso come:
$$SA[b], SA[b+1], \ldots, SA[e-1] \tag{5.40}$$
ovvero stiamo identificando le occorrenze esatte di Q nel testo.
- Data una BWT B, possiamo trovare i simboli che precedono i suffissi che condividono Q come prefisso come:
$$B[b], B[b+1], \ldots, B[e-1] \tag{5.41}$$
ovvero i simboli che precedono le occorrenze esatte di Q nel testo T.

Quindi possiamo dire che il numero di occorrenze di Q in T sarà $e-b$ se $[b,e) \equiv Q$- intervallo.

Esempio 22. *Dall'esempio precedente il aca-intervallo corrisponde a* $[4,6)$ *quindi saranno un totale di* $6-4=2$ *occorrenze di aca nel testo, iniziano alle posizioni* $1,5$ *del testo e sono precedute dai simbli* $\$, a$

Successivamente definiamo

Definizione 51 (**backward extension**). *La* ***backward extension*** *di un* Q*-intervalo* $[b,e)$ *con il carattere* σ *è il* σQ*-intervalo, cioè l'intervallo relativo alla stringa ottenuta concatenando il simbolo* σ *con* Q*.*

Esempio 23. *Riprendendo l'esempio precedente, dato il a-intervallo, vogliamo trovare la backward extension di a-intervallo col carattere c. Questo coincide con il gaca-intervallo ovvero* $[8,10)$*.*

Avendo definito questi concetti, possiamo definire l'algoritmo di ricerca esatta di un pattern P lungo m in un testo T utilizzando la BWT. Tale algoritmo è composto dai seguenti passi:

1. Si parte dal suffisso ϵ del pattern P.
2. Si considera il ϵ-intervallo, cioè $[1, n+1)$ e si inizializza un indice di posizione $i = m$.
3. Si effettua una backward extension con il simbolo $P[i]$ per ottenere $P[i]Q$-intervallo $[b_p, e_p)$ che fornisce le occorrenze di $P[i,m]$:
   - Se il risultato è l'intervallo vuoto allora P non ha occorrenze in T quindi si termina l'esecuzione.
   - Se il risultato non è l'intervallo vuoto allora:
     - se $i > 1$ allora si calcola $i--$ e si ripete il punto 3.
     - se $i = 1$ allora ho trovato il P-intervallo e si ritorna l'occorrenza.

Possiamo effettuare le seguenti osservazioni:

- Per capire se P occorre in T allora si hanno al massimo m iterazioni sul testo in cui calcoliamo il Q'-intervallo.
- Se occorre P in T allora servono un totale di m iterazioni, altrimenti saranno minori.
- Se P non occorre in T, supponendo che p è la posizione di P per cui il $P[p,m]$-intervallo è vuoto, allora l'ultimo suffisso che occorre in T è $P[p+1,m]$ in un numero di iterazioni $P[p+1,m]+1$
- La complessità quindi sarà $O(mlogn)$.

Possiamo rimuovere il $\log n$ nell'equazione del tempo rendendo costante il calcolo della backward extension. Possiamo osservare che per un Q-intervallo ($[b,e)$), e dato un simbolo σ:

- Siano $i_1, < i_2 < \cdots < i_k$ le k posizioni di $[b,e)$ tali che
$$B[i_q] = \sigma \ 1 \leq p \leq K \tag{5.42}$$
- $B[i_p]$ precede $S[i_p]$-suffisso per $1 \leq p \leq k$.
- $S[i_p]$-suffisso ha Q come prefisso per $1 \leq p \leq k$.
- Quindi $B[i_p] + S[i_p]$-suffisso per $1 \leq p \leq k$ è il suffisso che inizia con $B[i_p]$ e che ha σQ come prefisso.
- $B[i_p] + S[i_p]$-suffisso per LF-mapping avrà una posizione nel Suffix Array data da $j_p = LF(i_p)$ all'interno di $j_1, < j_2 < \cdots < j_k$, dove $[j_1, j_k)$ è il σQ-intervallo.

Da notare che, la backward extension viene definita dalle posizioni limite prese dal SA alla posizione BWT coincidente al carattere da cercare.

Quindi la procedura sarà definita come: Quindi ora si può mostrare l'algoritmo di ricerca esatta è: Algoritmo è lineare ($O(m)$) solo se definiamo il calcolo costante di LF.

Algorithm 10 Algoritmo per il calcolo della backward extension

function BACKWARD_EXTEND(b,e,σ)
  $i_1 \leftarrow \min_{x \in [b,e)} x$ t.c. $B[i_1] = \sigma$
  $i_k \leftarrow \max_{x \in [b,e)} x$ t.c. $B[i_k] = \sigma$
  return $[LF(i_1), LF(i_k) + 1)$
end function

Algorithm 11 Algoritmo di ricerca esatta del pattern nel testo

function SEARCH_PATTERN(P, T)
  $n \leftarrow |T|$
  $[b, e) \leftarrow [1, n + 1)$
  $i \leftarrow |P|$
  while $[b, e) \neq \text{null} \wedge i \geq 1$ **do**
    $\sigma \leftarrow P[i]$
    $[b, e) \leftarrow$ BACKWARD_EXTEND(b, e, σ)
    $i \leftarrow i - 1$
  end while
  if $[b, e) \neq \text{null}$ **then**
    out $T[b], T[b + 1], \ldots, T[e - 1]$
  end if
end function

5.6.3 FM-Index

Proposto da Ferragina e Manzini nel 2000, consiste in una rappresentazione della BWT del testo traite 2 funzioni numeriche. La ricerca esatta di un pattern P in un testo T è in $\Theta(|P|)$

L'FM-index è una struttura dati che permette di rappresentare una BWT in modo numerico.

Definizione 52 (**FM-index**). *Dato un testo T di lunghezza n, l'FM-index è una coppia di funzioni:*

- *C definita come:*
$$C : \Sigma \to \{0, 1, \ldots, n\} \tag{5.43}$$
dove $C(\sigma)$ rappresenta il numero di simboli della BWT che sono minori di σ. Conta quanti sono i suffissi che iniziano con un simbolo inferiore a σ e ritorna la posizione nell'ordine lessicografico dell'ultimo suffisso che inizia ocn un simbolo più piccolo di σ.

- *Occ definita come:*
$$Occ : \{0, 1, \ldots, n\} \times \Sigma \to \{0, 1, \ldots, n\} \tag{5.44}$$
dove $Occ(i, \sigma)$ rappresenta il numero di occorrenze del simbolo σ nella BWT fino alla posizione i $B[1, i-1]$. Quindi ogni riga rappresenta la distribuzione dei simboli nella BWT fino alla cella i.

Esempio 24. *Dato il testo $T = ggtcagtc\$$, possiamo definire la funzione C come:*

σ	$C(\sigma)$
\$	0
a	1
c	2
g	4
t	7

In generale, $C(\sigma)$ fornisce:

- il numero di suffissi che iniziano con un simbolo minore di σ.
- la massima posizione nel suffix array di un suffisso che inizia con un simbolo minore di σ.

Esempio 25. *Data la seguente BWT $BWT = cctt\$aggg$, possiamo definire la funzione Occ come: Copio la riga precedente modificando la colonna del carattere che leggiamo nella nuova posizione di BWT.*

$Occ(i, \sigma)$	\$	a	c	g	t
$Occ(0, \sigma)$	0	0	0	0	0
$Occ(1, \sigma)$	0	0	1	0	0
$Occ(2, \sigma)$	0	0	2	0	0
$Occ(3, \sigma)$	0	0	2	0	0
$Occ(4, \sigma)$	0	0	2	0	1
$Occ(5, \sigma)$	0	0	2	0	2
$Occ(6, \sigma)$	1	1	2	0	2
$Occ(7, \sigma)$	1	1	2	0	2
$Occ(8, \sigma)$	1	1	2	1	2
$Occ(9, \sigma)$	1	1	2	2	2
$Occ(10, \sigma)$	1	1	2	3	2

L'ultima riga della tabella ci dice la distribuzione dei simboli.

Esempio 26. *Possiamo quindi ottenere $c(\sigma)$ da $Occ(|BWT|, \sigma)$. Si sommano i valori delle colonne precedenti sulla riga.*

σ	$C(\sigma)$
\$	0
a	1
c	2
g	4
t	7

La funzione $Occ(i, \sigma)$ si può costruire considerando la riga precedente e incrementando il valore di $Occ(i-1, \sigma)$ se il simbolo in posizione i è uguale a σ. Avendo definito queste due funzioni possiamo definire il calcolo della

function BUILD-FMINDEX(B)
  $n \leftarrow |B|$
  $C \leftarrow$ array di dimensione $|\Sigma|$
  $Occ \leftarrow$ matrice di dimensione $n + 1 \times |\Sigma|$
  for σ **do**
    $Occ[1, \sigma] \leftarrow 0$
  end for
  for $i \leftarrow 1$ **to** n **do**
    for σ **in** Σ **do**
      $Occ[i + 1, \sigma] \leftarrow Occ[i, \sigma]$
    end for
    $Occ[i + 1, B[i]] \leftarrow Occ[i + 1, B[i]] + 1$
  end for
  $C[\$] \leftarrow 0$
  for σ **in** Σ **do**
    $\sigma' \leftarrow$ simbolo precedente di σ
    $C[\sigma] \leftarrow C[\sigma'] + Occ[n + 1, \sigma']$
  end for
  return C, Occ
end function

LF-function in tempo costante calcolando il numero di suffissi che iniziano con un carattere più piccolo di $B[i]$ con un il numero di simboli si $B[i]$ che occorrono nella BWT prima della posizione i.

$$j = LF(i) = C(B[i]) + Occ(i, B[i]) + 1 \tag{5.45}$$

Utilizzando l'FM-index possiamo definire il caloclo della backward extension in tempo costante. Questo ci permette di definire un algoritmo per la ricerca esatta in tempo lineare rispetto alla dimensione del pattern.

Vediamo ora come utilizzare l'FM-index per calcolare la **backward extension** in tempo costante. Supponiamo che:

- i_1: è la più piccola posizione di $[b, e)$ tale che $B[i_1] = \sigma$
- i_k: è la più grande posizione di $[b, e)$ tale che $B[i_k] = \sigma$

Quindi possiamo dire che il nuovo intervallo $[b', e')$ si ottiene come:

- $b' = LF(i_1) = C(\sigma) + Occ(i_1, \sigma) + 1$
- $e' = LF(i_k) + 1 = C(\sigma) + Occ(i_1, \sigma) + 1 + 1$

Però non è ancora costante perché dobbiamo trovare $i_1,\ i_k$.

Effettuiamo le seguenti osservazioni:

- In $B[b, i_1]$ non esistono simboli uguali a σ quindi il numero di simboli σ in $B[1, i_1 - 1]$ ($Occ(i_1, \sigma)$) è uguale al numero di simboli di σ in $B[1, b - 1]$ ($Occ(b, \sigma)$) allora:

$$Occ(i_1, \sigma) = Occ(b, \sigma) \tag{5.46}$$

  Quindi la ricerca di i_1 è costante.

- $Occ(i_k, \sigma)$ è il numero di σ in $B[1, i_k - 1]$, inoltre sappiamo che $B[i_k] = \sigma$. Questo significa che $B[i_k] = \sigma$, quindi implica che $Occ(i_k, \sigma) + 1 = Occ(i_k + 1, \sigma)$. Quindi $B[i_k + 1, e - 1]$ non esistono simboli uguali a σ allora il numero di simboli σ in $B[1, i_k]$ ($Occ(i_k + 1, \sigma)$) uguale al numero di simboli σ in $B[1, e - 1]$, possiamo quindi dire:

$$Occ(i_k + 1, \sigma) = Occ(e, \sigma) \tag{5.47}$$

Possiamo quindi esprimere il calcolo della backward extension come:

$$\begin{aligned} b' &= C(\sigma) + Occ(b, \sigma) + 1 \\ e' &= C(\sigma) + Occ(e, \sigma) + 1 \end{aligned} \tag{5.48}$$

Nota 14. *Se $b' = e'$ allora il Q-intervallo è vuoto.*

Vediamo ora l'algoritmo per calcolare la backward extension in tempo costante ($\mathcal{O}(1)$):

function BACKWARDEXTENSION(b, e, σ)
  $b' \leftarrow C(\sigma) + Occ(b, \sigma) + 1$
  $e' \leftarrow C(\sigma) + Occ(e, \sigma) + 1$
  return $[b', e')$
end function

Nota 15. *L'FM-Index è indipendente dal testo e BWT, inoltre, la struttura è reversibile e si può ricavare testo e BWT. Infatti da C ricaviamo il SA mentre da Occ ricaviamo la BWT, quindi da questi posso ricostruire il testo T.*

*Quindi l'FM-index è un **self-index**.*

Esempio 27. *Per dimostrare che l'FM-index è un self-index mostriamo un esempio di come da esso è possibile ricavare la BWT e di conseguenza il testo.*

0	0	0	0	0
0	0	1	0	0
0	0	2	0	0
0	0	2	0	1
0	0	2	0	2
1	0	2	0	2
1	1	2	0	2
1	1	2	1	2
1	1	2	2	2
1	1	2	3	2
\$	a	c	g	t

$$B = [c, c, t, t, \$, a, g, g, g] \tag{5.49}$$

L'algoritmo per ottenere la BWT è il seguente:

σ	C(σ)
\$	0
a	1
c	2
g	4
t	7

- *Si scorre la matrice associata alla Occ una riga alla volta.*
- *Per ogni riga si trova il valore che è stato incrementato rispetto alla riga precedente. Quel valore è il carattere in posizione i della BWT.*

Esempio 28. *Si consideri la funzione C riportata nella tabella 5.1. Dire se il b-intervallo, ovvero il Q-intervallo*

σ	C(σ)
\$	0
a	1
c	3
g	7
t	7

Tabella 5.1: Funzione C

con $Q = b$ è vuoto. Se non lo è, specificare il b-intervallo.

Sarà $[C(b)+1, C(k)+1)$ con k il carattere successivo al carattere b. Questo è vero perché C ritorno il numero di suffissi nell'ordine lessicografico che iniziano con un carattere minore di b.

Esempio 29. *Data la $BWT = accgt\$ac$ di un testo T, si richiede di specificare l'FM-index supponendo $\Sigma = \{a, c, g, t\}$. Calcolare poi tramite FM-index la posizione j nel Suffix Array del suffisso che inizia col terzo simbolo della BWT.*

Ricaviamo $C(\sigma)$ da $Occ(|BWT| + 1, \sigma)$ ovvero: Ricaviamo il $Occ(i, \sigma)$: Posizione j nel Suffix Array del

σ	C(σ)
\$	0
a	1
c	3
g	6
t	7

0	0	0	0	0
0	1	0	0	0
0	1	1	0	0
0	1	2	0	0
0	1	2	1	0
0	1	2	1	1
0	1	2	1	1
1	1	2	1	1
1	2	2	1	1
\$	a	c	g	t

suffisso che inizia con $B[3]$ si ottiene come:

$$j = LF(B[3]) = C(B[3]) + Occ(3, B[3]) + 1 \tag{5.50}$$

Esempio 30. *Data $BWT = t\$ccaacc$ di un testo e sapendo che c-intervallo è $[4, 8)$ specifica utilizzando B:*

- *quante volte la stringa cc occorre in T e specificare il cc-intervallo*

- *quante volte la stringa tc occorre in T e specificare il tc-intervallo*

In aggiunta indica quali sono i simboli che nel testo precedono le occorrenze di cc e le occorrenze di tc.

Per sapere se occorre cc basta prendere il c-intervallo e controllare all'interno se ci sono c, in questo caso 2. Per il caso tc no.

i simboli che recedono le occorrenze sono per entrambi a, ricavate dal SA alla posizione accociate alla ultima c.

Esempio 31. *Ho un testo per cui*

$$C = 0, 1, 4, 6, 6$$

Dire se

$$Occ(11, \sigma) = 1, 2, 3, 2, 2$$

è compatibile con C? No perché $C(c) = 3 \neq 4$ e $C(t) = 8 \neq 6$.

Dire quanti sono i simboli di g

Esempio 32. *Dalla faunzione Occ determinare $C = 0, 1, 3, 4, 8$ e determina $B[4] = g$.*

Ti può chiedere se esiste un $\sigma \cdot q$-intervallo guardando Occ.