Relazione di Laboratorio

Brian Riccardi

Indice

Parte 1

In questa prima parte studieremo il Problema proposto nel Laboratorio di Algoritmi e Strutture Dati 2016/17.
Proporremo una soluzione

$$\langle \Theta(|T| + |W|^2), \Theta(|T| + |W|^2) \rangle$$

per un testo T che contiene W parole distinte, analizzeremo la Complessità Computazionale sia per via analitica sia per via empirica e studieremo la relazione tra T e W per ottenere un'espressione della complessità solo in termini di $|T|$.

1.1 Il Problema

Il problema che vogliamo risolvere è la costruzione di un grafo $G = (W, R)$ sulle parole di un testo $T \in (\Sigma \setminus \{\not{\emptyset}\})^+$ e la ricerca della lunghezza del massimo cammino semplice (per numero di archi) in G.
La relazione R è definita da

$$R = \{(u, v) \in W \times W : \vec{u} \neq \vec{v} \wedge \forall i \; \vec{u}[i] \geq \vec{v}[i]\} \tag{1.1}$$

dove $\vec{u}$ e $\vec{v}$ sono i vettori delle occorrenze dei caratteri di Σ in u e v.

In parole povere, in G è presente un arco $u \rightsquigarrow v$ se e solo se la parola u contiene tutte le lettere [1] della parola v e $\vec{u} \neq \vec{v}$.

1.2 Convenzioni

Al fine di evitare incomprensioni e per alleggerire la notazione utilizzeremo le seguenti convenzioni, che saranno valide se non viene specificato altrimenti:

- $\sigma = |\Sigma|$, la cardinalità dell'alfabeto
- $t = |T|$, la lunghezza del testo
- $f_\sigma(x) = \log_\sigma \left(\frac{x}{\log_\sigma(x)} \right)$
- $\max \varnothing = 0$

[1] Tenendo conto anche delle ripetizioni.

Inoltre, ogni qualvolta ci riferiremo a testi o insiemi di parole daremo per scontato di aver fissato un alfabeto Σ.

1.3 Proprietà

Enunceremo ora due proprietà utili per le sezioni successive che dimostreremo nella sezione dedicata.

La prima proprietà descrive la topologia del grafo G.

Teorema. *Il grafo G è un DAG (Directed Acyclic Graph).*

La seconda mette in relazione il numero di parole distinte che posso trovare in un testo e la lunghezza del testo stesso.

Teorema. $|W| \in \mathcal{O}(\frac{t}{f_\sigma(t)})$.

1.4 La soluzione

L'idea è molto semplice:

1. Leggeremo l'input T
2. Riconosceremo Σ e costruiremo una mappa $alphabet : [0, 256) \to [0, \sigma)$ con lo scopo di restringere ASCII-extended nei soli caratteri presenti in T
3. Enumereremo le stringhe distinte di T
4. Dalle parole distinte costruiremo G
5. Troveremo il cammino massimo in G sfruttando le sue proprietà topologiche
6. Stamperemo il risultato della ricerca e il grafo stesso in formato *dot*

Di questi, i punti $3, 4, 5$ sono quelli a cui porremo più attenzione in questa relazione in quanto gli altri sono sufficientemente chiari dal codice.

1.4.1 Enumeriamo W

Una volta letto T vorremmo creare una lista W delle parole distinte.
L'idea è quella di scandire T con una macchina a due stati e, ogni volta che troveremo una parola w, capire se non è già stata scoperta ed eventualmente aggiungerla a W.

Ma come fare a decidere se w deve essere aggiunto a W?
Un approccio semplice potrebbe essere quello di confrontare w con tutte le parole già presenti in W. Tuttavia, come vedremo nelle prossime sezioni, con un algoritmo di questo tipo la costruzione di W avrebbe costo $\Omega((\frac{t}{f(t)})^2)$ al caso pessimo.

Per riuscire a migliorare i tempi costruiremo una struttura dati *Trie*, ovvero un albero σ-*ario* che utilizzeremo per salvare le stringhe di T.
Una stringa w viene salvata nel Trie scandendola da sinistra a destra e muovendosi nell'albero seguendo il figlio corrispondente alla lettera nella posizione di w che stiamo valutando. Aggiungeremo quindi un flag `true` ai nodi che sono la terminazione di una parola. In questo modo, quando, provando ad inserire w nel Trie, scopriremo che termina in un nodo `true` sapremo per certo che w è già stata trovata; in caso contrario la aggiungeremo a W.

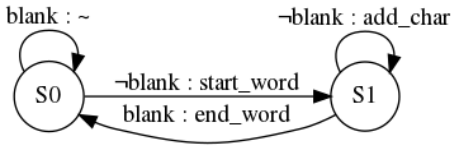

Figura 1.1: La macchina a stati per leggere le parole dal testo.

1.4.2 La costruzione di G

Avendo a disposizione W possiamo costruire G semplicemente guardando a tutte le $\frac{|W|(|W|-1)}{2}$ coppie di `vettori-occorrenze` e decidendo se e in che verso aggiungere l'arco soddisfando R.

1.4.3 La ricerca del cammino massimo

Una volta costruito il grafo non ci resta che ricercare la lunghezza del cammino massimo. Come abbiamo già enunciato in precedenza, G è un DAG e ammette un ordinamento topologico.
Allora sappiamo che, da ogni nodo, possiamo calcolare la lunghezza del cammino massimo che inizia in quel nodo e che quella lunghezza è un numero certamente finito (questo perché non ci sono cicli).
Sia quindi $length(u)$ la lunghezza del cammino massimo che inizia in u, allora vale la seguente [2]:

$$length(u) = \max_{v \in adj(u)} \{1 + length(v)\} \tag{1.2}$$

(a) (b)

Figura 1.2: (a) Un Trie che conserva le parole *no*, *nu* e *r*. (b) Lo stesso Trie dopo aver inserito la parola *n*. I nodi rossi indicano il flag `true` settato.

[2] Non è altro che una definizione ricorsiva sui figli di u, con caso base implicito su $\max \varnothing$.

Costruiremo quindi un ordinamento topologico di G e scandiremo i nodi in ordine, dai pozzi alle sorgenti.
Ad ogni passo, mentre valuteremo il nodo u, applicheremo l'equazione (2) e useremo una variabile `MAX` per mantenere il massimo assoluto delle lunghezze.

1.5 Analisi della complessità computazionale

Rifacendoci all'elenco dell'idea risolutiva possiamo ricavare la complessità della soluzione come somma delle complessità dei singoli sotto-algoritmi. Come prima, porremo maggiore attenzione ai soli punti $3, 4$ e 5: gli altri sono già molto chiari dai commenti riportati sul codice.

1. La lettura dell'input ha costo

$$\Theta(|T|)$$

2. La costruzione della mappa ha costo

$$\Theta(|T|)$$

3. L'enumerazione di W ha costo

$$\Theta(|T|)$$

   infatti la macchina a stati legge carattere per carattere il testo T e, appena completa una parola w, questa viene inserita nel Trie al costo di $\Theta(|w|)$; in base all'esito dell'inserimento spenderemo un ulteriore $\mathcal{O}(|w|)$ per costruirne l'eventuale `vettore-occorrenze` e inserirla in W. Questo procedimento lo ripetiamo per tutte le parole di T ed essendo

$$\sum_{w \in T} |w| \leq |T|$$

   otterremo il costo stimato.
   (Da non dimenticare che leggiamo tutto T, quindi il costo è $\Omega(|T|)$)

4. La costruzione di G ha costo

$$\Theta(|W|^2) \subseteq \mathcal{O}\Big(\Big(\frac{t}{f_\sigma(t)}\Big)^2\Big)$$

   infatti vengono confrontate $\frac{|W|(|W|-1)}{2}$ coppie di `vettori-occorrenze` e ogni coppia viene confrontata in tempo $\Theta(\sigma) = \Theta(1)$; in tutto $\Theta(\sigma|W|^2) = \Theta(|W|^2)$; l'inclusione deriva dal **Teorema 2**

5. La ricerca del percorso massimo ha costo

$$\Theta(|W| + |R|) \subseteq \mathcal{O}(|W|^2) \subseteq \mathcal{O}\Big(\Big(\frac{t}{f_\sigma(t)}\Big)^2\Big)$$

   infatti l'ordinamento topologico ha costo lineare nelle dimensioni del grafo, quindi $\Theta(|W|+|R|)$; la ricerca del valore di `MAX` è un semplice ciclo for che rivisita ogni arco e ogni nodo al costo $\Theta(|W| + |R|)$; R è relazione binaria su W quindi $|R| \in \mathcal{O}(|W|^2)$ e quindi

$$\Theta(|W| + |R|) \subseteq \mathcal{O}(|W|)^2$$

   L'ultima inclusione deriva dal **Teorema 2**

6. La stampa dei risultati e del grafo in formato *dot* ha costo

$$\Theta(|W| + |R|) \subseteq \mathcal{O}(|W|^2) \subseteq \mathcal{O}\Big(\Big(\frac{t}{f_\sigma(t)}\Big)^2\Big)$$

Il costo totale dell'algoritmo è dato dalla somma dei costi dei singoli punti. La soluzione ha complessità in tempo

$$\Theta(|T| + |W|^2 + |R|) = \Theta(|T| + |W|^2) \subseteq \mathcal{O}\Big(\Big(\frac{t}{f_\sigma(t)}\Big)^2\Big)$$

1.6 Scelte implementative e organizzazione

L'algoritmo proposto è stato implementato nel linguaggio C++ sfruttandone gli standard C++14.

1.6.1 Memoria e codice

La soluzione descritta funziona molto bene quando ragioniamo ad alto livello; tuttavia, per implementarla, dobbiamo considerare diversi fattori aggiuntivi tra cui la chiarezza del codice e, soprattutto, il consumo effettivo (e non asintotico) di memoria.

Caratteri o interi?
La gestione di testi diventa una pugnalata al cuore quando si introducono gli ASCII-extended, quindi ho deciso di manipolare i caratteri dell'input come se fossero gli interi associati agli ASCII degli stessi. Nella pratica non cambia praticamente nulla ma ho voluto scriverlo lo stesso per evitare fraintendimenti quando vengono comparati l'idea risolutiva e il codice effettivo.
(Input e Output sono comunque dei veri e propri testi e non insiemi di numeri).

La classe std::vector.
Il C++ mette a disposizione nelle sue librerie standard una classe che astrae un array ridimensionabile.
Il motivo per l'utilizzo di questa classe riguarda principalmente il metodo `push_back`: questi permette l'inserimento in coda al vector di un nuovo elemento in tempo $\Theta(1)$ (mantenendo l'accesso diretto in tempo costante a tutti gli elementi del vector)[3]. Avendo a disposizione un metodo di questo tipo non devo preoccuparmi di contare (ad esempio) la lunghezza di testi e stringhe prima di allocare la memoria. È certamente un aiuto e probabilmente avrei dovuto implementarlo da solo tuttavia, considerandolo solo un modo per favorire la chiarezza del codice più che una struttura dati indispensabile alla risoluzione del problema, ho preferito utilizzarlo.

[3]Come? L'idea è di iniziare allocando un array `a` con una quantità definita di memoria; ad ogni `push_back`, se c'è spazio sufficiente inserisco l'elemento, altrimenti alloco un nuovo array `b` con il doppio della memoria e ne copio tutti gli elementi di `a` e il nuovo elemento. In questo modo, l'inserimento di n elementi prevede $\lfloor \log_2(n) \rfloor$ riallocazioni e un costo totale di $\Theta(1 + 2 + 4 + \cdots + 2^{\lfloor \log_2(n) \rfloor}) = \Theta(n)$ e quindi un costo medio per `push_back` di $\Theta(1)$.
Fonte del costo

Quanti figli nel Trie?
Nel problema l'alfabeto potrebbe essere tutto ASCII-extended, ciò significa che ogni nodo del Trie avrebbe, alla peggio, 256 figli oltre alla variabile `flag`; ogni nodo peserebbe ben $(256 + 1) \cdot 4$ Byte (più di 1KB!) e dato che un Trie alla peggio ha (circa) t nodi si avrebbe un consumo esagerato di memoria.
Per migliorare la situazione ho deciso di guardare alle parole non solo come array di numeri ASCII bensì come lunghe stringhe di bit (convertendo e giustapponendo gli ASCII).
Ad esempio: `NO` $\rightsquigarrow$ 78 79 $\rightsquigarrow$ 1001110 1001111 $\rightsquigarrow$ 10011101001111.

In questo modo, non considerando gli spazi, un testo di lunghezza t diventerebbe di lunghezza $8t$ ma si avrebbe una diminuzione di figli da 256 a 2 e quindi da un consumo di $t \cdot (256 + 1) \cdot 4$ Byte a $8t \cdot (2 + 1) \cdot 4$ Byte, circa un decimo!
Per inserire w nel Trie necessitiamo di un modo rapido per ottenere $b_i(w)$ (l'i-esimo bit di w). L'idea è quella di isolare innanzitutto la lettera in cui si trova $b_i(w)$, applicare un `AND` bit a bit (& rubando la notazione C/C++) per isolare il singolo bit e infine shiftare a destra il tutto.

$$b_i(w) = (w\big[i \text{ div } 8\big] \ \& \ 2^{7-(i \bmod 8)}) \text{ div } (2^{7-(i \bmod 8)}) \tag{1.3}$$

1.6.2 Organizzazione della cartella

La cartella `Progetto` è stata organizzata in questo modo

graph/ contiene degli output personali già in formato *dot* e *png*

input/ è la destinazione dei Campioni generati durante le misurazioni, all'interno sono presenti degli input personali

text/ contiene questa relazione in formato *pdf* e *tex* ed eventuali file prodotti dalla compilazione di LaTeX

lib/ contiene dei sorgenti personali che ho utilizzato come librerie:

- asd_gen è il codice dei generatori
- asd_trie è la struttura Trie
- asd_util implementa i `PRNG`
- asd_word è la struttura dati che astrae una coppia (`parola`, `vettore_occorrenze`)

misurazioni/ contiene i programmi per i test

output/ la destinazione degli output dei Campioni di misura

sol/ contiene il programma per testare la correttezza e la soluzione del progetto

util/ contiene un file di testo con dei parametri che verranno descritti successivamente

- *seed*
- c_n
- l
- ab

1.6.3 Compilazione e run

Testing della correttezza
Per compilare il programma che verrà utilizzato per testarne la correttezza posizionarsi nella cartella `Progetto/sol/` e lanciare

```
g++ -std=c++14 -O2 sol.cpp problem.cpp -o sol
```

E per lanciarlo semplicemente

```
./sol
```

Misura dei tempi
Per compilare il programma della misura dei tempi posizionarsi nella cartella `Progetto/misurazioni` e lanciare

```
g++ -std=c++14 -O2 misura.cpp ../sol/problem.cpp -o misura
```

E per lanciarlo semplicemente

```
./misura ordine
```

Dove `ordine` è un numero 10^k a scelta.

1.7 Analisi sperimentale

In questa sezione descriveremo gli esperimenti effettuati per ricavare una stima ragionevole dei tempi d'esecuzione e confronteremo i risultati sperimentali con i risultati analitici.

1.7.1 tMin, rip, tare, ecc...

Gli appunti messi a disposizione coprono diverse problematiche.
In C/C++ è presente una funzione `clock` che ritorna il numero di tic della CPU dall'inizio del processo; è anche presente una costante `CLOCKS_PER_SEC` che rappresenta il numero di tic al secondo nell'hardware corrente (nel nostro caso 10^6).
Ciò significa che le mie misurazioni non necessitano del calcolo di `tMin` e `rip` in quanto il mio orologio è virtualmente perfetto.
Grazie a queste considerazioni posso aggiungere alla funzione `problem`, ovvero la soluzione del progetto, un semplice calcolo del tempo di esecuzione tramite una differenza di `clock` tra l'inizio e la fine dell'esecuzione.
In questo modo, il codice per il calcolo dei tempi risulta enormemente più semplice. Per quanto riguarda l'`algoritmo9`, è stato lasciato praticamente invariato.

1.7.2 I generatori di test

I tre generatori sfruttano l'`algoritmo8` per produrre numeri pseudo-casuali.
Inoltre, fanno riferimento a parametri c_n, l, a e b presenti in un file collocato in una cartella precisa. In particolare, c_n rappresenta la cardinalità del Campione di misura: di fatto, ogni generatore è pensato per costruire un Campione di c_n testi lunghi uguale.

gen_randomcase(t) genera c_n testi lunghi t caratteri e composti da stringhe lunghe l generate casualmente su di un alfabeto ASCII$\big[a, b\big)$.

gen_bestcase(t) genera c_n testi lunghi t caratteri e composti da $\sqrt{t}$ stringhe lunghe l generate casualmente su di un alfabeto ASCII$\big[a, b\big)$. La scelta di $\sqrt{t}$ è da ricercarsi nella complessità al caso peggiore della soluzione: costruendo $\sqrt{t}$ stringhe ho la possibilità di avere svariati test diversi ma tutti con una soluzione in $\Theta(t)$.

gen_worstcase(t) genera c_n testi lunghi t caratteri e composti da sempre le stesse stringhe (in ordine sparso). L'idea è quella di ottenere sempre il caso pessimo del mio algoritmo, ovvero quando il numero di parole distinte è massimo. Infatti le stringhe costruite da `gen_worstcase` sono tutte le stringhe binarie dalla lunghezza 1 fino a una lunghezza massima che dipende da t.

1.7.3 Procedimento di misurazione

Lanciando il programma `misura` e passando (da linea di comando) un parametro 10^k verranno prodotti dei Campioni da ogni generatore. Questi Campioni saranno su testi di lunghezze $t \in \{10, 20, 30, 50, 100, 200, \cdots, 3 \cdot 10^k, 5 \cdot 10^k\}$.

Ogni Campione viene misurato dalla funzione `measure_time` che esegue l'`algoritmo9`; i risultati vengono poi stampati su dei file di testo.
Per tutti i test sono stati utilizzati i valori

- $c_n = 10$
- $l = 15$
- $a = 48, b = 58$
- $\alpha = 0.02$
- $z_\alpha = 2.33$
- $\Delta = \frac{t^2}{const}$ per una costante *const*, su `randomcase` e `worstcase`
- $\Delta = \frac{t}{const}$ per una costante *const*, su `bestcase`

La macchina su cui sono stati effettuati i test ha le seguenti specifiche

Processore Intel i5-4210U, 4 core, 1.70 GHz

RAM 8 Gb

Sistema Operativo Korora Linux 26, 64bit

1.7.4 Risultati sperimentali

I test sono stati effettuati lanciando `$./misura 100000` e hanno prodotto dei risultati assolutamente in linea con l'analisi analitica della complessità.

I grafici arancioni (che rappresentano la complessità teorica) sono stati scalati di una costante che rappresenta, circa, i secondi necessari per l'esecuzione di 1 istruzione semplice.

In particolare

- **bestcase** ha complessità $\Theta(t)$ in quanto la procedura dominante è la lettura e la manipolazione del testo.
- **worstcase** ha complessità $\Theta((\frac{t}{f_\sigma(t)})^2)$
- **randomcase** ha la complessità del caso pessimo in quanto, per come sono stati costruiti i test, la probabilità di ottenere almeno due parole uguali è piccolissima.[4]

L'analisi sperimentale conferma quindi l'analisi analitica effettuata nelle sezioni precedenti.

Bestcase			
Input	Misura	Delta all'uscita	Complessità Teorica
10	0,000011	0,000002	0,000001
20	0,000029	0,000005	0,000001
30	0,000028	0,000002	0,000002
50	0,000042	0,000004	0,000003
100	0,000079	0,000007	0,000005
200	0,000078	0,000004	0,000010
300	0,000103	0,000005	0,000015
500	0,000167	0,000024	0,000025
1000	0,000295	0,000041	0,000050
2000	0,000396	0,000081	0,000100
3000	0,000432	0,000061	0,000150
5000	0,000559	0,000066	0,000250
10000	0,001011	0,000176	0,000500
20000	0,001374	0,000045	0,001000
30000	0,002809	0,000562	0,001500
50000	0,003185	0,000311	0,002500
100000	0,005343	0,000261	0,005000
200000	0,010211	0,000445	0,010000
300000	0,015285	0,000314	0,015000
500000	0,025248	0,000594	0,025000

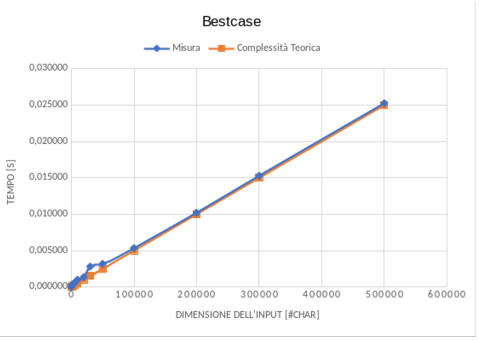

Figura 1.3: Campioni generati da **gen_bestcase**

[4] Ad essere sinceri non ho dei risultati completi, ma l'idea è che due parole sono identiche con probabilità $(\frac{1}{\sigma})^l = (\frac{1}{10})^{15} = 10^{-15}$, ovvero quando costruisco una parola, per ottenerne una uguale devo costruirne in media altre 10^{15} e nei testi più grossi ho solo (circa) 30000 parole.

Worstcase			
Input	Misura	Delta all'uscita	Complessità Teorica
10	0,000016	0,000001	0,000002
20	0,000022	0,000001	0,000004
30	0,000037	0,000003	0,000007
50	0,000051	0,000006	0,000013
100	0,000122	0,000005	0,000033
200	0,000258	0,000024	0,000090
300	0,000345	0,000063	0,000167
500	0,000576	0,000040	0,000371
1000	0,001568	0,000231	0,001131
2000	0,004082	0,000267	0,003545
3000	0,008748	0,001671	0,006995
5000	0,017937	0,000785	0,016635
10000	0,052988	0,000704	0,054758
20000	0,176461	0,001661	0,183111
30000	0,396691	0,022199	0,373387
50000	0,870679	0,024582	0,921744
100000	2,77476	0,015133	3,171675
200000	9,91961	0,030829	11,020669
300000	21,97393	0,117877	22,929155
500000	53,56528	0,203104	57,934148

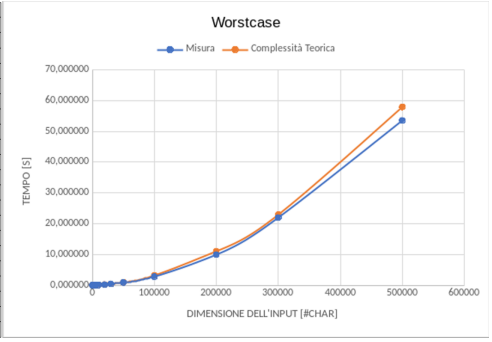

Figura 1.4: Campioni generati da **gen_worstcase**

Randomcase			
Input	Misura	Delta all'uscita	Complessità Teorica
10	0,000034	0,000005	0,000002
20	0,000032	0,000002	0,000004
30	0,000041	0,000002	0,000007
50	0,000064	0,000002	0,000013
100	0,000095	0,000016	0,000033
200	0,000131	0,000003	0,000090
300	0,000187	0,000003	0,000167
500	0,000308	0,000009	0,000371
1000	0,000669	0,000028	0,001131
2000	0,001776	0,000113	0,003545
3000	0,002953	0,000146	0,006995
5000	0,006552	0,000258	0,016635
10000	0,021152	0,000733	0,054758
20000	0,077405	0,000523	0,183111
30000	0,168453	0,000357	0,373387
50000	0,467088	0,011951	0,921744
100000	1,908270	0,158562	3,171675
200000	9,878360	0,969140	11,020669
300000	23,967005	0,589172	22,929155
500000	66,562870	0,559432	57,934148

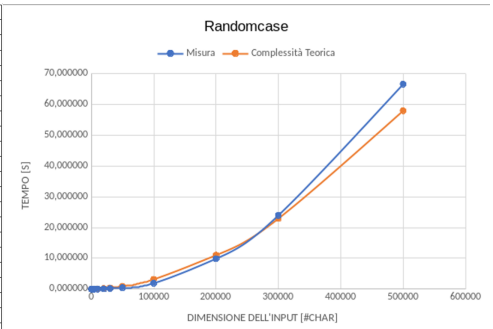

Figura 1.5: Campioni generati da **gen_randomcase**

1.8 Dimostrazioni

In questa sezione dimostreremo i risultati sfruttati nelle sezioni precedenti. Enunceremo e dimostreremo anche altri lemmi utili.

Lemma. *R è una relazione d'ordine stretto.*

Dimostrazione. Dimostriamo che R è **transitiva** e **antisimmetrica**.
(T) Siano $a, b, c \in W$ tali che $aRb \wedge bRc$. Dalla transitività di $\geq_{\mathbb{Z}}$ e dalla definizione di R deduco

$$\forall i \big(\vec{a}[i] \geq \vec{b}[i] \geq \vec{c}[i]\big) \wedge \vec{a} \neq \vec{b} \wedge \vec{b} \neq \vec{c}$$

Non può essere $\vec{a} = \vec{c}$, infatti se così fosse si avrebbe $\vec{a} = \vec{b} \wedge \vec{b} = \vec{c}$ contraddicendo $aRb \wedge bRc$ [5]. Ma allora aRc.

(A) Supponiamo $a, b \in W$ e, senza perdita di generalità, aRb. Dalla definizione di R deduco

$$\forall i \big(\vec{a}[i] \geq \vec{b}[i]\big) \wedge \vec{a} \neq \vec{b}$$

Se fosse bRa si avrebbe forzatamente $\vec{a} = \vec{b}$ contraddicendo aRb.

□

[5] $\vec{a}[i]$ e $\vec{c}[i]$ chiudono $\vec{b}[i]$ in un sandwich.

Teorema. *G è un DAG (Directed Acyclic Graph).*

Dimostrazione. G è il grafo che rappresenta l'insieme parzialmente ordinato (W, R). Se G ammettesse un ciclo, sia esso $v_1 v_2 \cdots v_k v_1$, per la **transitività** di R si avrebbe $v_1 R v_k \wedge v_k R v_1$ contraddicendo l'**antisimmetria**. □

Lemma.

$$\sum_{k=0}^{n} k\sigma^k \sim_\infty \frac{\sigma}{\sigma - 1} n\sigma^n$$

Dimostrazione.

$$\begin{aligned}
\sum_{k=0}^{n} k\sigma^k &= \sigma^1 + 2\sigma^2 + 3\sigma^3 + \cdots + n\sigma^n = \sigma(1 + 2\sigma + 3\sigma^2 + \cdots + n\sigma^{n-1}) \\
&= \sigma \sum_{k=1}^{n} \frac{d}{d\sigma}(\sigma^k) = \sigma \frac{d}{d\sigma}\Big(\sum_{k=1}^{n} \sigma^k\Big) = \sigma \frac{d}{d\sigma}\Big(\frac{\sigma^{n+1} - 1}{\sigma - 1} - 1\Big) \\
&= \sigma \frac{d}{d\sigma}\Big(\frac{\sigma^{n+1} - \sigma}{\sigma - 1}\Big) = \frac{\sigma}{(\sigma - 1)^2}\Big(n\sigma^{n+1} - (n+1)\sigma^n + 1\Big) \\
&= \frac{\sigma}{(\sigma - 1)^2} n\sigma^n \Big(\sigma - \frac{n+1}{n} + \frac{1}{n\sigma^n}\Big) \longrightarrow \frac{\sigma}{\sigma - 1} n\sigma^n
\end{aligned}$$

Quando $n \longrightarrow \infty$. □

Lemma. [6]

$$f_\sigma(t)\sigma^{f_\sigma(t)} \sim_\infty t$$

Dimostrazione.

$$\begin{aligned}
&\lim_{t\to+\infty} \frac{\log_\sigma\left(\frac{t}{\log_\sigma(t)}\right)\sigma^{\log_\sigma\left(\frac{t}{\log_\sigma(t)}\right)}}{t} = \lim_{t\to+\infty} \frac{\log_\sigma\left(\frac{t}{\log_\sigma(t)}\right)\frac{t}{\log_\sigma(t)}}{t} \\
&= \lim_{t\to+\infty} \frac{\log_\sigma\left(\frac{t}{\log_\sigma(t)}\right)}{\log_\sigma(t)} = \lim_{t\to+\infty} \frac{\log_\sigma(t) - \log_\sigma(\log_\sigma(t))}{\log_\sigma(t)} \\
&= \lim_{t\to+\infty} \frac{\cancel{\log_\sigma(t)}}{\cancel{\log_\sigma(t)}} \frac{1 - \frac{\log_\sigma(\log_\sigma(t))}{\log_\sigma(t)}}{1} = 1
\end{aligned}$$

□

Teorema. *Sia N_t il massimo numero di stringhe distinte di un testo lungo t. Allora*

$$N_t \sim_\infty \frac{t}{f_\sigma(t)}$$

Dimostrazione. L'idea è di costruire tutte le σ parole di lunghezza 1, tutte le σ^2 parole di lunghezza 2, tutte le σ^3 parole di lunghezza 3, ecc... Senza perdità

[6] f ricavabile con un pò di creatività e una notte in bianco passata a sbattere la testa su un muro

di generalità, assumiamo t valido per costruire quella sequenza di stringhe fino ad una lunghezza n (che dipende da t). Allora

$$t = 1\sigma^1 + 2\sigma^2 + 3\sigma^3 + \cdots + n\sigma^n = \sum_{k=1}^{n} k\sigma^k = \sum_{k=0}^{n} k\sigma^k$$

Quando $t \longrightarrow \infty$ anche $n \longrightarrow \infty$ e per i **lemmi** precedenti

$$t \sim_\infty \frac{\sigma}{\sigma - 1} n\sigma^n$$

Sia ora $t' = t\frac{\sigma-1}{\sigma}$, allora per il **lemma** precedente

$$n = f_\sigma(t')$$

Per concludere, un testo costruito in questo modo ha

$$N_t = \sum_{k=1}^{n} \sigma^k = \frac{\sigma^{n+1} - 1}{\sigma - 1} \sim_\infty \frac{\sigma}{\sigma - 1}\sigma^n$$

e mettendo insieme i risultati ottengo

$$N_t \sim_\infty \frac{t}{n} = \frac{t}{f_\sigma(t')} \sim_\infty^{7} \frac{t}{f_\sigma(t)}$$

□

Corollario.

$$|W| \in \mathcal{O}\Big(\frac{t}{f_\sigma(t)}\Big)$$

Dimostrazione.

$$|W| \leq N_t \sim_\infty \frac{t}{f_\sigma(t)}$$

□

[7] Il limite si risolve più o meno allo stesso modo del **lemma** precedente.

Parte 2

La **transitività** di R ci costringe ad avere un grafo molto connesso e nel peggiore dei casi il numero di archi è il quadrato del numero di vertici. Sappiamo perciò che la completa soluzione del Problema (che prevede la stampa di G) sarà sempre, al caso peggiore, quadratica nel numero delle parole.

Ma se volessimo solo la lunghezza del cammino? È ragionevole supporre che esista una soluzione lineare nella lunghezza del testo in quanto il cammino che cerchiamo non potrà mai essere più lungo del numero totale di parole e il numero di parole non potrà mai essere più grande della lunghezza del testo. Questo ci spinge ad indagare più a fondo sulla ricerca di quel cammino.

In questa seconda parte studieremo il Problema della sola ricerca della lunghezza del cammino massimo e proporremo un Algoritmo e una Struttura Dati per risolverlo in

$$\langle \mathcal{O}(|T| + |W| \log_2^\sigma |W|), \mathcal{O}(|T| + |W| \log_2^\sigma |W|) \rangle$$

asintoticamente migliore della soluzione proposta nella **Parte 1** ma nella pratica inutilizzabile a causa delle enormi costanti nascoste.

2.1 L'idea

Se non ci interessa la stampa di G perché impegnarci a costruirlo?

Ricordandoci dell'equazione (2) che definisce $length(u)$ notiamo che non fa riferimento a G ma solo ai figli di u in G; se fossimo in grado di trovare in fretta $length(u)$ senza costruire G avremo un algoritmo migliore rispetto a quello proposto nello scorso capitolo.

Enunciamo ora due proprietà di cui ne omettiamo (le banali) dimostrazioni.

Osservazione. *Se ordinassimo lessicograficamente i* `vettori_occorrenze`*, ogni vettore avrebbe discendenti solo tra gli elementi alla sua sinistra.*

Osservazione. *Se disegnassimo i* `vettori_occorrenze` *in* $\mathbb{R}^\sigma$ *i discendenti di* u *si troverebbero tutti nella regione*

$$\mathcal{R} = \big[0, \vec{u}[1]\big] \times \cdots \times \big[0, \vec{u}[\sigma]\big]$$

Immaginiamo ora di possedere un algoritmo `query(`u`, `L`)` che in tempo $\mathcal{O}(\log_2^\sigma |W|)$ calcola il valore $length(u)$ quando $L[v] = length(v)$ per i discendenti v di u. Allora un possibile algoritmo per il calcolo di `MAX` nella complessità enunciata potrebbe essere:

Algorithm 1 GetMaxLength(W)

```
1: MAX ← 0
2: L ←NewArray(|W|, 0)          ▷ Array lungo |W| popolato di soli 0
3:
4: Sort(W)
5: for all u ∈ W do
6:     L[u] ← query(u,L)
7:     MAX ← max(MAX, L[u])
8: return MAX
```

Dalla prima osservazione sappiamo per certo che ad ogni passo del ciclo i discendenti di u sono già stati visitati e quindi L possiede la proprietà di cui parlavamo sopra.
Il nostro obbiettivo quindi è scrivere `query`.

2.1.1 Divide et Impera

Immaginiamo la famiglia di relazioni