Laboratorio di ASD - prima parte

Bazzana Lorenzo, 147569
bazzana.lorenzo@spes.uniud.it

D'Ambrosi Denis, 147681
dambrosi.denis@spes.uniud.it

Zanolin Lorenzo, 148199
zanolin.lorenzo@spes.uniud.it

24 Marzo 2021

1 Introduzione

Richiesta È stato richiesto di implementare ed analizzare la complessità di due algoritmi per il calcolo del *periodo frazionario minimo*.

Il periodo frazionario minimo di una stringa s di lunghezza $|s| = n$ è definito come il più piccolo intero $p > 0$ tale che

$$\forall i = 1, ..., n - p \quad s(i) = s(i + p)$$

I due algoritmi discussi all'interno di questa relazione per la risoluzione del problema prendono il nome di *PeriodNaive* e *PeriodSmart*: come si vedrà nei prossimi paragrafi, le due procedure implementano metodi nettamente diversi nel calcolo del periodo frazionario minimo e questa eterogeneità si rifletterà in tempi di esecuzione molto differenti tra le due funzioni.

2 Presentazione degli algoritmi

2.1 Algoritmi per il calcolo del periodo frazionario minimo

PeriodNaive La procedura prende in input una stringa e, attraverso un ciclo che itera al più n volte, verifica per valori di p crescenti se la sottostringa $s[1, p]$ è periodo frazionario minimo s con $O(n - p)$ confronti a ogni ciclo. È evidente, quindi, che la complessità nel caso pessimo di questo algoritmo è $O(n^2)$.

PeriodSmart Definito il *bordo* di una stringa s come una sottostringa di s che è sia un suo prefisso proprio che un suo suffisso proprio, si può notare che un periodo frazionario $s[1, p]$ è tale se e solo se $p = n - r$, in cui r è la lunghezza di un bordo di s. Di conseguenza, determinare la lunghezza del

periodo frazionario minimo di una stringa equivale a trovare la lunghezza del suo bordo massimo (per poi, ovviamente, sottrarla alla lunghezza della stringa stessa). La funzione *PeriodSmart* calcola induttivamente la lunghezza del bordo massimo delle sottostringhe $s[1, i], i \in [1, n]$. Applicando la memoization per rendere tale procedura efficiente, è possibile ottenere una complessità dell'ordine di $\Theta(n)$.

2.2 Algoritmi per la generazione di stringhe

Per analizzare l'andamento degli algoritmi sopra descritti è essenziale generare stringhe di lunghezza data in maniera pseudocasuale. Per questo scopo è necessario determinare la sequenza di lunghezze n e una serie di procedure capaci di creare stringhe con caratteristiche differenti in base alla tipologia di performance (caso medio, caso pessimo, ecc...) che si vuole andare ad analizzare.

Nel nostro caso, le stringhe generate hanno lunghezza compresa nell'intervallo $[1000, 500000]$ e sono distribuite secondo una funzione esponenziale in i del tipo

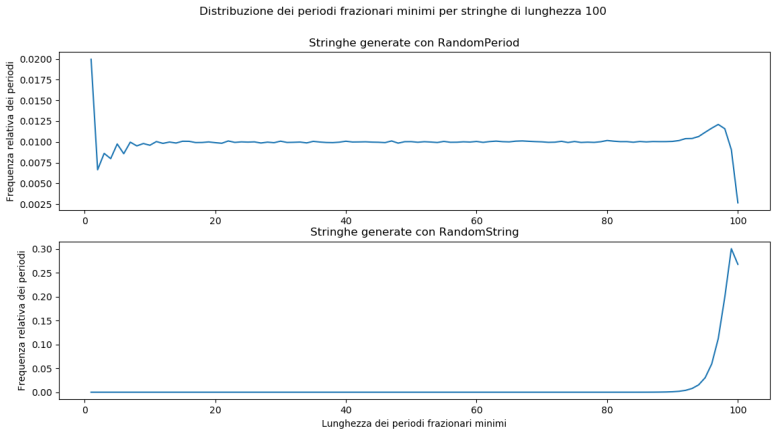
$$n_i = \lfloor AB^i \rfloor, i = 1, 2, ..., 100, A = 1000, B = e^{\frac{\log 500000 - \log A}{99}} \approx 1.064785978$$

Tale funzione permette di ottenere un alto numero di stringhe brevi: in tal modo è possibile analizzare stringhe con lunghezze appartenenti diversi ordini di grandezza evitando, almeno in parte, il problema dato dalle computazioni troppo lunghe.

RandomString La procedura costruisce una stringa di lunghezza prefissata n, accodando caratteri estratti casualmente nell'insieme $\{a, b\}$.

RandomPeriod La procedura genera, richiamando *RandomString*, un periodo di lunghezza casuale $q \in [1, n]$ e poi ne accoda $\frac{n}{q}$ ripetizioni (eventualmente troncando in fondo).

Questi due algoritmi sono stati usati nella maggior parte delle misurazioni effettuate e, in quanto tali, è opportuno spendere qualche parola in più per analizzare le differenze di output tra i due.



Come si può notare, *RandomString* tende a generare stringhe con periodi frazionari minimi molto più lunghi: è statisticamente improbabile, infatti, che la stessa (breve) sequenza di caratteri venga composta molte volte nell'arco nelle n estrazioni necessarie a comporre l'output. Al contrario, *RandomPeriod* garantisce una distribuzione quasi uniforme di periodi nell'intervallo $[1, n]$ in quanto sceglie casualmente la lunghezza della sequenza da replicare all'interno del suddetto intervallo prima di comporre la stringa. È interessante notare, tuttavia, che la distribuzione dei periodi non è propriamente discreta uniforme: è possibile (seppur molto poco probabile, come si può evincere dal grafico) che un periodo di lunghezza p venga generato casualmente con un "sottoperiodo" di lunghezza compresa in $[1, p]$. Ad esempio, la funzione *random* richiamata in *RandomPeriod* per determinare la lunghezza del periodo potrebbe fornire come output il valore 5. Se *RandomString*, richiamata con tale input, fornisse in output "*aaaaa*", è facilmente intuibile come anche *RandomPeriod* restituirebbe una stringa con periodo frazionario minimo $p = 1$, seppur fosse stato estratto casualmente il valore 5.

BestPeriod La procedura costruisce una stringa di n caratteri uguali al fine di fornire un periodo frazionario minimo di $p = 1$ caratteri.

WorstPeriod La procedura genera una stringa di $n - 1$ caratteri uguali, a cui accoda un'occorrenza del carattere complementare. In tal modo l'output presenta un periodo frazionario minimo di $p = n$ caratteri.

3 Misurazione dei tempi di esecuzione

La misurazione dei tempi necessari a calcolare il periodo frazionario viene effettuata tenendo conto della lunghezza n della stringa s_n e dell'errore commesso: in particolare, per ogni data lunghezza della stringa, l'algoritmo implementato ripete il calcolo del periodo un numero di volte tale da assicurare un errore massimo relativo pari a $\epsilon_{max} = 0.001$, garantito da un tempo totale maggiore o uguale a T_{min}, calcolato come

$$T_{min} = R(\frac{1}{\epsilon} + 1)$$

in cui R è la risoluzione del clock.

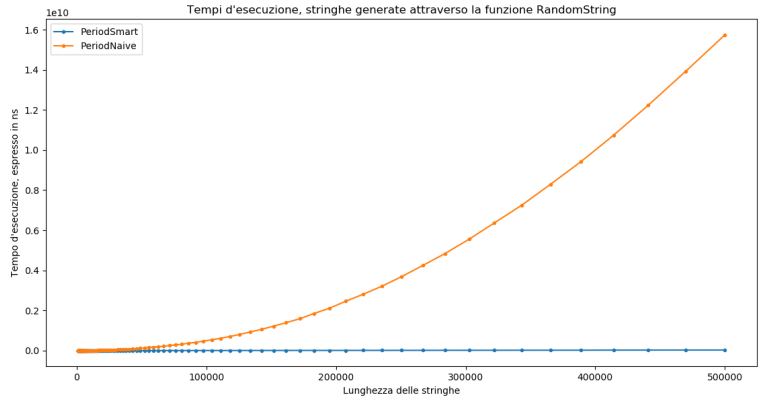
Per ogni lunghezza fissata ciascuna iterazione del calcolo del periodo viene effettuata su una stringa generata pseudo-casualmente, diversa quindi da iterazione a iterazione. Ciò comporta che nel tempo totale registrato viene incluso anche quello impiegato a generare le stringhe. Per questo motivo si determina il tempo medio necessario a calcolare il periodo di stringhe s_n come:

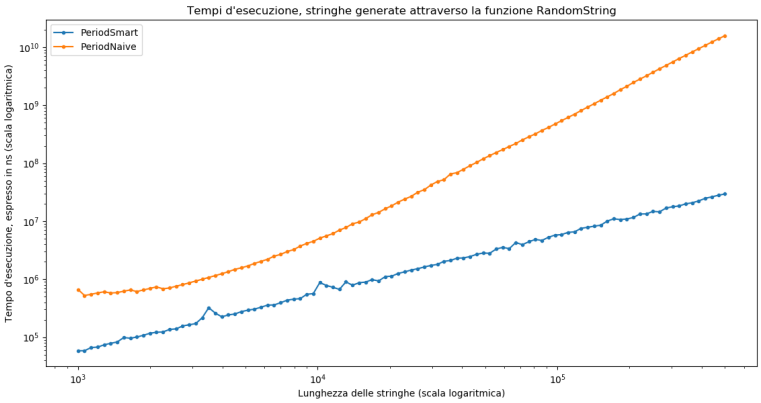
$$T_n = \frac{tempo\ totale - tempo\ generazione\ stringhe}{k}$$

dove k è il numero di iterazioni necessarie a garantire un errore massimo pari a ϵ_{max}.

4 Rapprensentazione grafica dei risultati

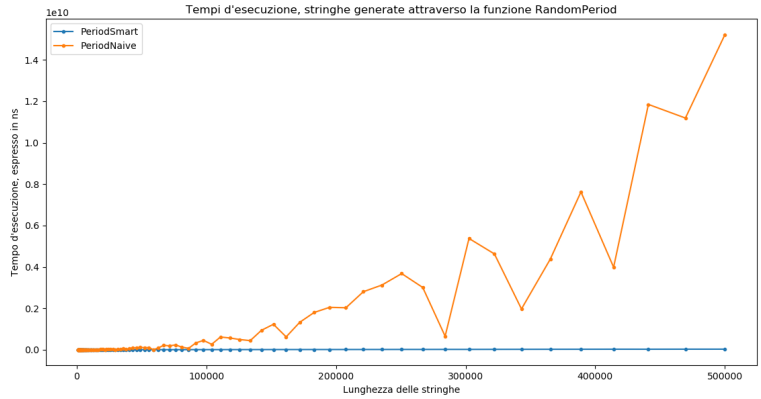
4.1 Stringhe generate con *RandomString*

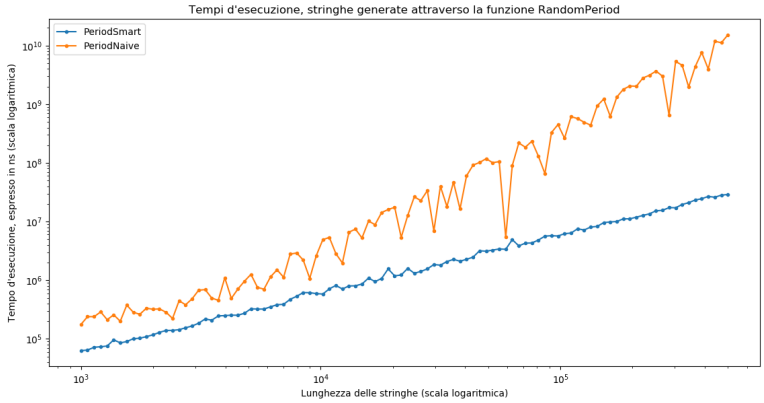




Come si evince dai grafici, *PeriodNaive* presenta un andamento quadratico che si evidenzia all'aumentare della lunghezza della stringa s. Al contrario, *PeriodSmart* garantisce performance superiori grazie alla sua complessità lineare, confermata dall'andamento rettilineo del suo grafico in scala doppiamente lineare.

4.2 Stringhe generate con *RandomPeriod*

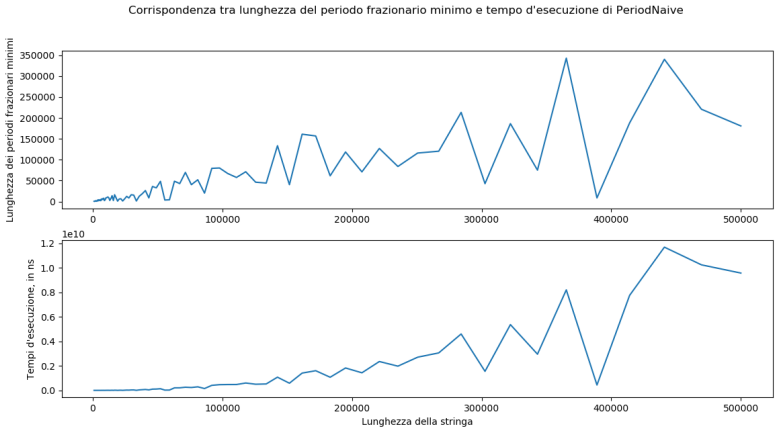




È interessante notare come i tempi d'esecuzione dell'algoritmo *PeriodNaive* ottenuti fornendo in input stringhe generate con la procedura *RandomPeriod* siano più imprevedibili di quelli misurati nelle iterazioni che utilizzavano le stringhe create da *RandomString*. Ciò è dovuto al fatto che *RandomString*, a differenza dell'algoritmo *RandomPeriod*, genera stringhe con periodi frazionari statisticamente molto lunghi, come spiegato nella sezione 2. Se infatti la procedura *PeriodSmart* presenta andamento lineare a prescindere dalla stringa fornita (a conferma della complessità $\Theta(n)$ accennata nella sezione Presentazione degli algoritmi), l'algoritmo *PeriodNaive* in certi casi impiega tempi drasticamente inferiori a quelli attesi nell'andamento quadratico. Come verrà dimostrato successivamente, infatti, la complessità $O(n^2)$ rappresenta solo un limite superiore alle performance di tale procedura: con periodi frazionari brevi anche questo algoritmo può dimostrarsi efficiente. A conferma di ciò, è interessante osservare il seguente grafico che mette a confronto la lunghezza dei periodi frazionari generati attraverso *RandomPeriod* con i tempi d'esecuzioni dell'algoritmo *PeriodNaive*

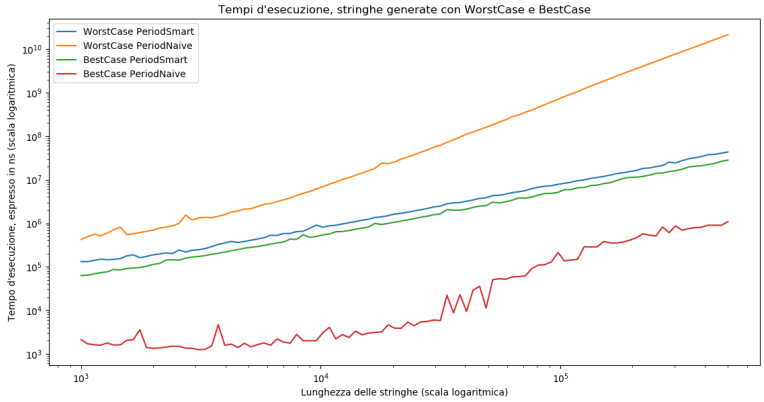
per le relative stringhe.

4.2.1 Confronto tra la lunghezza dei periodi generati e i tempi d'esecuzione di *PeriodNaive*



La corrispondenza biunivoca tra periodi brevi e performance migliori è dovuta alla struttura dell'algoritmo *PeriodNaive*: ad ogni iterazione del ciclo verifica se la sottostringa $s[1, n-p]$ e la sottostringa $s[p, n]$ coincidono, con $O(n-p)$ confronti di caratteri. Se il periodo frazionario minimo è breve, la condizione viene verificata in poche iterazioni del ciclo: nel caso in cui $p \in \Theta(1)$, allora l'algoritmo assume complessità lineare!

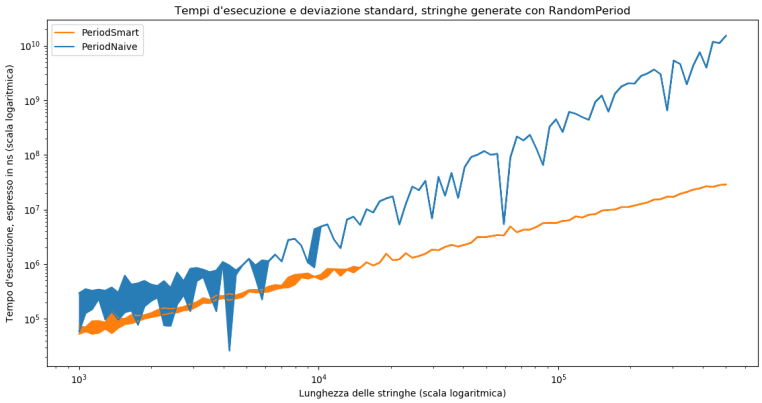
4.3 Stringhe generate con *WorstCase* e *BestCase*



Il grafico soprastante mostra infatti come le performance di *PeriodNaive* siano estremamente eterogenee nel caso pessimo (ossia in cui $p = n$) rispetto al caso ottimo (ossia in cui $p = 1$), mentre l'algoritmo *PeriodSmart* garantisce sempre

tempi di esecuzione relativamente indipendenti dalla lunghezza del periodo frazionario minimo.

4.4 Visualizzazione della deviazione



Mettendo a grafico le due funzioni e la deviazione standard per ogni lunghezza n si può notare come l'indice di variabilità sia mediamente minore nel caso dell'algoritmo *PeriodSmart* rispetto a *PeriodNaive*. Questo comportamento indica che nei vari cicli per ogni lunghezza fissata il primo algoritmo ha avuto un comportamento più omogeneo rispetto a *PeriodNaive*; bisogna anche considerare che ad ogni iterazione la computazione avviene su una nuova stringa generata pseudo-casualmente, quindi il periodo può variare molto da stringa a stringa e di conseguenza anche il tempo richiesto da *PeriodNaive*. Una cosa interessante da notare è che per entrambi gli algoritmi, a partire da stringhe molto lunghe, la deviazione standard vale 0: questo accade perché il tempo necessario alla computazione di una singola stringa è abbastanza elevato da non richiedere più di una iterazione per ciclo, ovvero rientra nell'intervallo di tempi ritenuto accettabile.