Periodo frazionario

Bozza

2023-08-17

# Periodo frazionario

## Scopo del progetto

Lo scopo di questo progetto è implementare due algoritmi per il calcolo del *periodo frazionario minimo* di una stringa e misurarne i tempi medi di esecuzione. Per *periodo frazionario minimo* si intende, data una stringa *S*, la lunghezza (*p*) della sottostringa più breve che può essere ripetuta in modo da formare la stringa *S*. Deve quindi soddisfare la seguente proprietà:

* : stringa
* : lunghezza del periodo
* : lunghezza della stringa
* : parte esterna al periodo frazionario

## Algoritmi e fondamenti teorici

### PeriodNaive

*PeriodNaive* è un algoritmo di complessità che, mediante un ciclo *for*, itera su di una stringa, e ne analizza ogni possibile periodo. Partendo da un periodo di lunghezza *p=1* e incrementando di uno ogni volta il valore, la stringa viene suddivisa in due parti: *head* e *tail*. Quando queste coincidono l’esecuzione termina e viene restituita la lunghezza finale *p*.

Funzione periodoNaive(Stringa):  
 Length= lunghezza(Stringa)  
 ITERA P da 1 fino Length{  
 Head= sottostringa(Stringa, 0 -> Length-P )  
 Hail= sottostringa(Stringa, P -> Length)  
  
 SE Head è uguale a Tail:  
 RESTITUISCI P  
 }

#### Calcolo di correttezza

#### Calcolo di complessità

| Codice | Costo | Volte |
| --- | --- | --- |
| public static int periodNaive(String stringa) { |  |  |
| int n= stringa.length(); |  | 1 |
| for (int p=1; p<=n;p++){ |  |  |
| String head=stringa.substring(0, n-p); |  | n |
| String tail=stringa.substring(p, n); |  | n |
| if (head.equals(tail)){ |  | n |
| return p; |  | 1 |
| } |  |  |
| } |  |  |
| return ERR; |  |  |
| } |  |  |

Dalla tabella risulta che il costo computazionale dell’algoritmo è:

### PeriodSmart

*PeriodSmart* è un algoritmo di complessità che utilizza il concetto di *bordo* per calcolare la lunghezza del *periodo frazionario minimo*. Genera un *array* per memorizzare la lunghezza dei bordi delle sottostringhe (r[i] è il bordo della sottostringa che termina in posizione i - 1) e poi *itera* sulla stringa cercando per ogni carattere un bordo fra i caratteri precedenti. Se il carattere corrente coincide col bordo, incrementa la lunghezza del bordo, altrimenti la riduce cercando un bordo più corto. Il periodo frazionario minimo è dato dalla lunghezza della stringa meno la lunghezza massima del bordo.

Funzione periodSmart(S: stringa):  
 N = lunghezza(s)  
 R = array di interi di dimensione n + 1  
 R[1] = 0  
 ITERA I da 2 fino a N:  
 J = R[I - 1]  
 FINCHE J > 0,  
 carattere in (i - 1) di S != dal carattere in posizione J di S:  
 J = R[J]  
 SE carattere in (I - 1) di S == al carattere in J di s:  
 R[I] = J + 1  
 ALTRIMENTI:  
 R[I] = 0  
 MaxBordo = R[N]  
 PeriodoFrazionario = N - MaxBordo  
 RESTITUISCI PeriodoFrazionario

#### Calcolo di correttezza

#### Calcolo di complessità

| Codice | Costo | Volte |
| --- | --- | --- |
| public static int periodSmart(String s) { |  |  |
| int n = s.length(); |  | 1 |
| int[] r = new int[n + 1]; |  | 1 |
| r[1] = 0; |  | 1 |
| for (int i = 2; i <= n; i++) { |  |  |
| int j = r[i - 1]; |  | n |
| while (j > 0 && s.charAt(i - 1) != s.charAt(j)) { |  | n |
| j = r[j]; |  | n |
| } |  |  |
| if (s.charAt(i - 1) == s.charAt(j)) { |  | n |
| r[i] = j + 1; |  | n |
| } else { |  |  |
| r[i] = 0; |  | n |
| }} |  |  |
| int maxBordo = r[n]; |  | 1 |
| int periodoFrazionario = n - maxBordo; |  | 1 |
| return periodoFrazionario; |  | 1 |
| } |  |  |

Dalla tabella risulta che il costo computazionale dell’algoritmo è:

## Analisi delle prestazioni

Per valutare le prestazioni degli algoritmi in maniera empirica si è deciso di realizzare un programma Java in grado di misurare i tempi medi di esecuzione degli algoritmi. Il progetto si suddivide in 5 classi:

* **Progetto**: classe principale che contiene i passaggi per testare, cronometrare e ricavare dati utili in fase di analisi.
* **Algos**: contiene l’implementazione degli algoritmi per il calcolo del periodo frazionario minimo di una stringa.
* **Chrono**: fornisce meccanismi di cronometraggio necessari per misurare il tempo di esecuzione del programma e la stima di risoluzione del clock di sistema in nanosecondi.
* **Logger**: gestisce la registrazione delle informazioni rilevanti su un file .csv.
* **StringGenerator**: genera stringhe casuali con varie lunghezze comprese fra 1000 e 500000. Queste hanno una distribuzione esponenziale e sono basate su un alfabeto ternario: *a*, *b*, *c*.

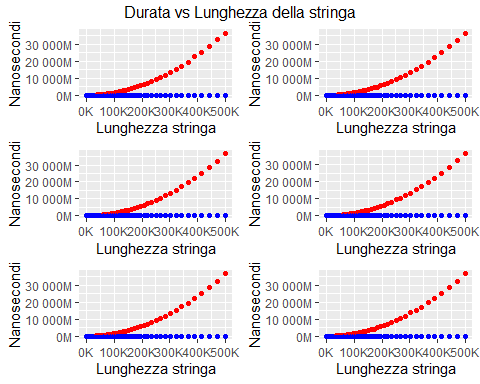
### Metodologia di test

Una volta calcolato il *tempo minimo misurabile*, per ricavare dati attendibili sono state eseguite 6 run da 100 iterazioni ciascuna. Ad ogni iterazione una stringa generata casualmente, e progressivamente sempre più lunga, è stata fornita in input ad entrambi gli algoritmi. È stato utilizzato un ciclo *while* per iterare l’esecuzione dell’algoritmo e misurare il tempo trascorso, fino a quando tale tempo non è risultato superiore al *tempo minimo misurabile*. Il *tempo medio di esecuzione* per una singola istanza di input è stato calcolato come il rapporto fra il tempo trascorso e il numero di iterazioni “*while*” eseguite.

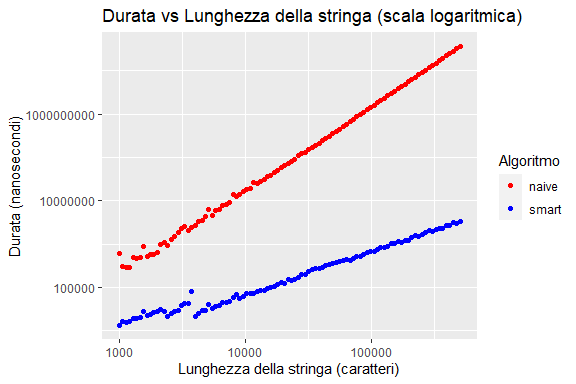
Progetto main():  
 t\_min= tempo\_minimo\_misurabile()  
 ITERA Run da 1 a 5{  
 ITERA Tedt da 0 a 99{  
 stringa= generatringa()  
 //testa algoritmo 1  
 iterazione= 0  
 start= now()  
 FAI{  
 algoritmo(stringa)  
 end= now()  
 iterazione++  
 }FINCHÈ (end-start< tmin)  
 tempo\_medio= (end-start)/i  
 // testa algoritmo 2  
 ...  
 }  
 }

### Risultati sperimentali di vari test

Analizzando i grafici *Durata* vs *Lunghezza della stringa* si nota come, i tempi di risoluzione relativi all’algoritmo *PeriodNaive* crescono esponenzialmente, mentre, quelli relativi a *PeriodSmart* hanno una crescita lineare.

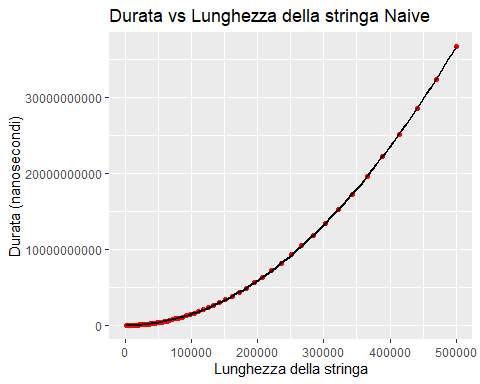


Sin dalle prime iterazioni il delta temporale fra i due algoritmi è dell’ordine di qualche millisecondo. Questa questa differenza aumenta esponenzialmente raggiungendo oltre mezzo secondo nelle iterazioni finali.



Basandosi sui dati raccolti, è possibile creare un modello in grado di prevedere l’andamento temporale dei due algoritmi. In particolare l’evoluzione del modello Naive può essere descritta da un’equazione di secondo grado:

La validità del modello è supportata dagli indici statistici, in particolare il *Residual Standard Error* indica come l’errore standard dei residui sia di circa 13.73 millisecondi. Gli indici *Multiple R-squared* e *Adjusted R-squared*, atti a spiegare la variabilità dei dati,hanno totalizzato il valore massimo di 1. Infine il *p-value* è di gran lunga inferiore alla soglia di 0.05 (), il che indica un risultato statisticamente significativo.



L’evoluzione del modello Smart può essere, invece, descritta da un’equazione di primo grado:

Anche in questo caso la validità del modello è supportata dagli indici statistici, in particolare il *Residual Standard Error* indica come l’errore standard dei residui sia di circa 0,0448 millisecondi. Gli indici *Multiple R-squared* e *Adjusted R-squared*,hanno totalizzato il valore quasi massimo di 0.9969. Infine il *p-value* è di gran lunga inferiore alla soglia di 0.05 (), il che indica un risultato statisticamente significativo.

