

Laboratorio di Algoritmi e Strutture Dati

2020/2021 — Prima parte

Mattia Bonaccorsi, Enrico Stefanel, Andriy Torchanyyn

9 marzo 2021

1 Il periodo frazionario

Definizione Il *periodo frazionario minimo* di una stringa s è il più piccolo intero $p > 0$ che soddisfa la seguente proprietà:

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

dove n denota la lunghezza e $s(k)$ è il k -esimo carattere della stringa s .

Esempio Il *periodo frazionario minimo* della stringa `abcbcab` è 3, mentre il *periodo frazionario minimo* della stringa `aba` è 2.

Si implementano di seguito due algoritmi scritti in linguaggio Python3 per il calcolo del *periodo frazionario minimo* di una qualsiasi stringa.

In particolare, l'algoritmo *PeriodNaïve* con complessità $\Theta(n^2)$, e l'algoritmo *PeriodSmart* con complessità $\Theta(n)$.

2 Algoritmo *PeriodNaïve*

2.1 Descrizione dell'algoritmo

L'algoritmo utilizza un ciclo `for` con un intero p che varia da 1 a n e termina restituendo p alla prima iterazione che soddisfa la proprietà (\star) . Quest'ultima viene verificata confrontando la congruenza tra il prefisso della stringa s fino alla posizione $n - p$, e il suffisso dalla posizione p fino al termine della stringa. Nel caso in cui queste parti combacino, la dimensione p è esattamente il *periodo frazionario minimo* cercato.

Nel caso dell'inserimento in input della stringa vuota, l'algoritmo restituirà il risultato di errore -1 in quanto n varrà 0 e quindi il codice all'interno del ciclo `for` non verrà mai eseguito.

2.2 Codice dell'algoritmo

```
1 # Returns the minimum fractional period of s
2 def PeriodNaive(s):
3     n = len(s)
4     for p in range(1,n+1):
5         if(_isPeriod(s,p)):
6             return p
7     return -1 # The input is an empty string
8
9 # Returns True if p is a fractional period of s,
10 # False otherwise.
11 def _isPeriod(s,p):
12     n = len(s)
13     if(s[:n-p] == s[p:]):
14         return True
15     else:
16         return False
```

2.3 Osservazioni sul codice

Alternativamente al controllo della congruenza tra il prefisso $[1 \dots n-p]$ e suffisso $[p \dots n]$, la funzione *isPeriod* potrebbe venire implementata anche con un ciclo secondario per il controllo dell'uguaglianza tra $s(j)$ e $s(j+p)$, con j che varia da 1 a $n-p$.

Tale implementazione sarebbe stata scritta così:

```
1 # Returns True if p is a fractional period of s,
2 # False otherwise.
3 def _isPeriod_2(s,p):
4     n = len(s)
5     for j in range(0,n-p):
6         if(s[j] != s[j+p]):
7             return False
8     return True
```

Il motivo per cui non è stata scelta questa implementazione è che nel caso di un generico input s , il ciclo interno si sarebbe fermato alla prima disuguaglianza trovata. Questo avrebbe di certo ottimizzato l'algoritmo in pratica, ma avrebbe reso più difficile un confronto generale tra questo algoritmo e la versione *smart* illustrata di seguito, che ha un andamento lineare sulla dimensione dell'input.

3 Algoritmo *PeriodSmart*

3.1 Descrizione dell'algoritmo

Definizione Un *bordo* di una stringa s è una qualunque stringa t che sia, allo stesso tempo, prefisso proprio di s e suffisso proprio di s .

Si osserva che p è un periodo frazionario di s se e solo se $s = |p| - r$, dove r è la lunghezza di un bordo di s . Ciò permette di ridurre il problema del calcolo del periodo frazionario minimo di s al problema del calcolo della lunghezza massima di un bordo di s .

Per risolvere quest'ultimo problema si procede per induzione, calcolando per ogni prefisso $s[1, \dots, i]$, dal più corto al più lungo, la lunghezza $r(i)$ del bordo massimo di $s[1, \dots, i]$. Per implementare il passo induttivo da i a $i + 1$ si consideri la sequenza

$$r(i) > r(r(i)) > r(r(r(i))) > \dots > r^k(i) = 0$$

e si osserva che nel calcolo di $r(i + 1)$ solamente i due casi seguenti possono darsi:

- per qualche indice $j \leq k$ vale l'uguaglianza $s[i + 1] = s[r^j(i) + 1]$. In tal caso, $r(i + 1) = r^j(i) + 1$ dove j è il primo indice per cui vale la suddetta uguaglianza;
- non esiste alcun indice $j \leq k$ che soddisfi l'uguaglianza $s[i + 1] = s[r^j(i) + 1]$. In tal caso, $r(i + 1) = 0$.

3.2 Codice dell'algoritmo

```
1 # Example of Theta(1) function
2 def PeriodSmart(s):
3     a = 'a' * 10_000_000
4     return True
```

4 Calcolo dei tempi di esecuzione dei due algoritmi

Si vogliono misurare i tempi medi di esecuzione dei due algoritmi *PeriodNaïve* e *PeriodSmart* al variare della lunghezza n della stringa fornita in input.

4.1 Calcolo della precisione del sistema

La precisione della misurazione sperimentale è data dal più piccolo intervallo di tempo che il nostro computer può misurare.

Possiamo stimare la risoluzione del *clock* di sistema utilizzando un ciclo `while` per calcolare l'intervallo minimo di tempo misurabile.

Successivamente si calcola il *tempo minimo ammissibile* in funzione della risoluzione stimata e dell'errore relativo ammissibile (pari, nel nostro caso, a 0.001).

```
1 # System time resolution
2 import time as t
3
4 TIME_ERROR = 0.001
```

```

5
6 start = t.time()
7 end = t.time()
8
9 while (start == end):
10     end = t.time()
11
12 r = end - start
13 t_min = r*((1/TIME_ERROR)+1)

```

```

1 print(t_min)
2 0.0009546279907226562

```

Il *tempo minimo ammissibile*, nel calcolatore utilizzato, è di 0.0009546279907226562 secondi.

4.2 Generazione dell'*array* delle dimensioni degli input

Si definisce un *array sizes* di lunghezze n dell'input con distribuzione esponenziale per ottenere un buon compromesso in termini di efficienza e di completezza dello studio. Tale *array* viene definito come l'insieme di tutti i valori $\lfloor A \cdot B^i \rfloor$, con $0 \leq i < 100$ e A, B calcolati opportunamente in modo da avere $n = 1000$ quando $i = 0$, e $n = 500000$ quando $i = 99$.

$$\begin{aligned}
 \lfloor A \cdot B^i \rfloor = 1000 \quad \text{se } i = 0 & \Rightarrow \lfloor A \cdot 1 \rfloor = 1000 & \Rightarrow A = 1000 \\
 \lfloor 1000 \cdot B^i \rfloor = 500000 \quad \text{se } i = 99 & \Rightarrow \lfloor B^{99} \rfloor = 500 & \Rightarrow B = \sqrt[99]{500}
 \end{aligned}$$

```

1 # Set choosen parameters
2 FIRST_SIZE = 1_000
3 LAST_SIZE = 500_000
4 RANGE = 100
5
6 # Create an array with exponential sizes
7 # in range(FIRST_SIZE, LAST_SIZE)
8 sizes = []
9 a = FIRST_SIZE
10 b = (LAST_SIZE/FIRST_SIZE)**(float(1/(RANGE-1)))
11 for i in range(RANGE):
12     sizes.append(int(a*(b**i)))

```

4.3 Generazione dell'*array* degli input

La stringa s di lunghezza n viene generata su un alfabeto ternario $\{a, b, c\}$ in modo che ogni lettera $s(i)$ della stringa sia generata in modo pseudo-casuale indipendentemente dalle altre.

```

1 # with individual lengths from sizes array
2 strings = []
3 for size in sizes:
4     strings.append(''.join(
5         random.choices(['a','b','c'], k=size)
6     ))

```

4.4 Calcolo del tempo medio

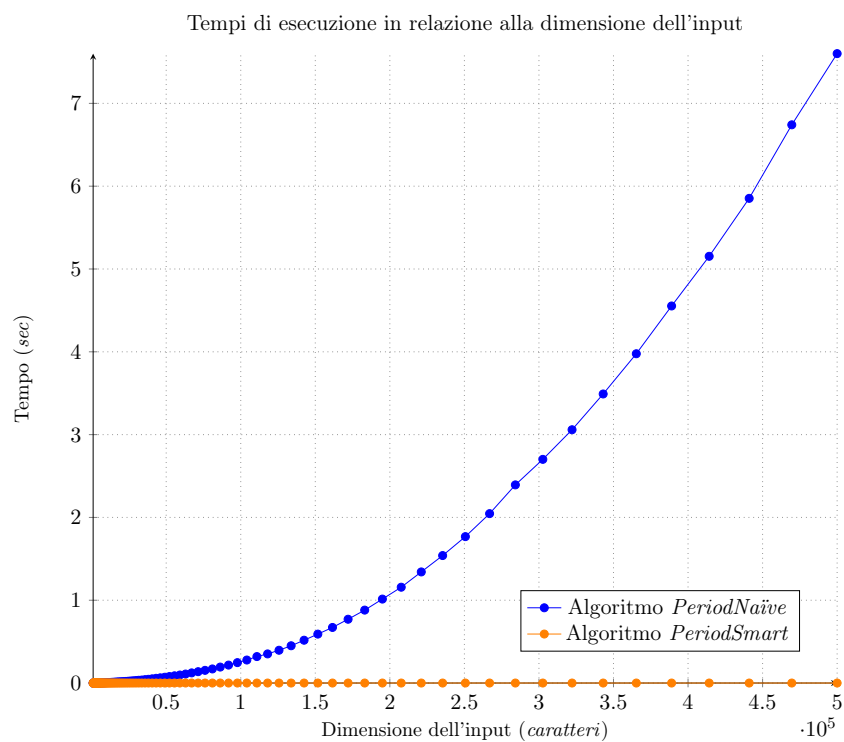
Vengono salvati in due array `growth_PN` e `growth_PS` rispettivamente i tempi di esecuzione dell'algoritmo *PeriodNaïve* e *PeriodSmart* al variare della lunghezza n della stringa fornita in input, seguendo la distribuzione indicata dall'array `strings` calcolata in precedenza.

```

1 # Create an array with timings from PeriodNaive function
2 # with input from strings array
3 growth_PN = []
4 PeriodNaive('')
5 for s in strings:
6     start = t.time()
7     PeriodNaive(s)
8     end = t.time()
9     growth_PN.append(end-start)
10
11 # Create an array with timings from PeriodSmart function
12 # with input from strings array
13 growth_PS = []
14 PeriodSmart('')
15 for s in strings:
16     start = t.time()
17     PeriodSmart(s)

```

Visualizzando i risultati dei tempi di esecuzione con le rispettive lunghezze dell'input, si ottiene il grafico sotto.



Per poter apprezzare meglio le misurazioni, è possibile visualizzare il grafico con scala logaritmica su entrambi gli assi.

