

Laboratorio di Algoritmi e Strutture Dati

2020/2021 — Prima parte

Mattia Bonaccorsi — 124610 – bonaccorsi.mattia@spes.uniud.it

Muhammed Kouate — 137359 – kouate.muhammed@spes.uniud.it

Enrico Stefanel — 137411 – stefanel.enrico@spes.uniud.it

Andriy Torchanyyn — 139535 – torchanyyn.andriy@spes.uniud.it

8 aprile 2021

Indice

1	Il periodo frazionario	2
2	Algoritmo <i>PeriodNaïve</i>	2
2.1	Descrizione dell'algoritmo	2
2.2	Codice dell'algoritmo	2
2.2.1	Osservazioni sul codice	3
3	Algoritmo <i>PeriodSmart</i>	3
3.1	Descrizione dell'algoritmo	3
3.2	Codice dell'algoritmo	4
4	Calcolo dei tempi di esecuzione dei due algoritmi	4
4.1	Calcolo della precisione del sistema	4
4.2	Generazione dell' <i>array</i> delle dimensioni degli input	5
4.3	Generazione dell' <i>array</i> degli input	6
4.4	Calcolo del tempo medio	7
4.5	Risultati della sperimentazione	8
5	Ulteriori osservazioni	9
5.1	Caso pessimo per l'algoritmo <i>PeriodNaïve</i>	9
5.1.1	Ideazione dell'input	9
5.2	Codice dell'algoritmo	10
5.2.1	Risultati della sperimentazione	11

1 Il periodo frazionario

Definizione Il *periodo frazionario minimo* di una stringa s è il più piccolo intero $p > 0$ che soddisfa la 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 `abcabcab` 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 $O(n^2)$, 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.

2.2 Codice dell'algoritmo

```
1  # Returns the minimum fractional period of s
2  def PeriodNaive(s):
3      assert s, "String can't be empty!"
4      n = len(s)
5      for p in range(1, n+1):
6          if(_isPeriod(s, p)):
7              return p
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     return (s[:n-p] == s[p:])
```

sources/PeriodNaive.py

2.2.1 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
```

sources/PeriodNaive.py

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 # Returns the minimum fractional period of s
2 def PeriodSmart(s):
3     n = len(s)
4     pf = [0]
5     for i in range(1, n):
6         j = pf[i-1]
7         while ((j > 0) and (s[i] != s[j])):
8             j = pf[j-1]
9         if (s[i] == s[j]):
10             j += 1
11         pf.append(j)
12     return n - pf[n-1]
```

sources/PeriodSmart.py

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)
```

sources/Complexity.py

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 $i = 99$.

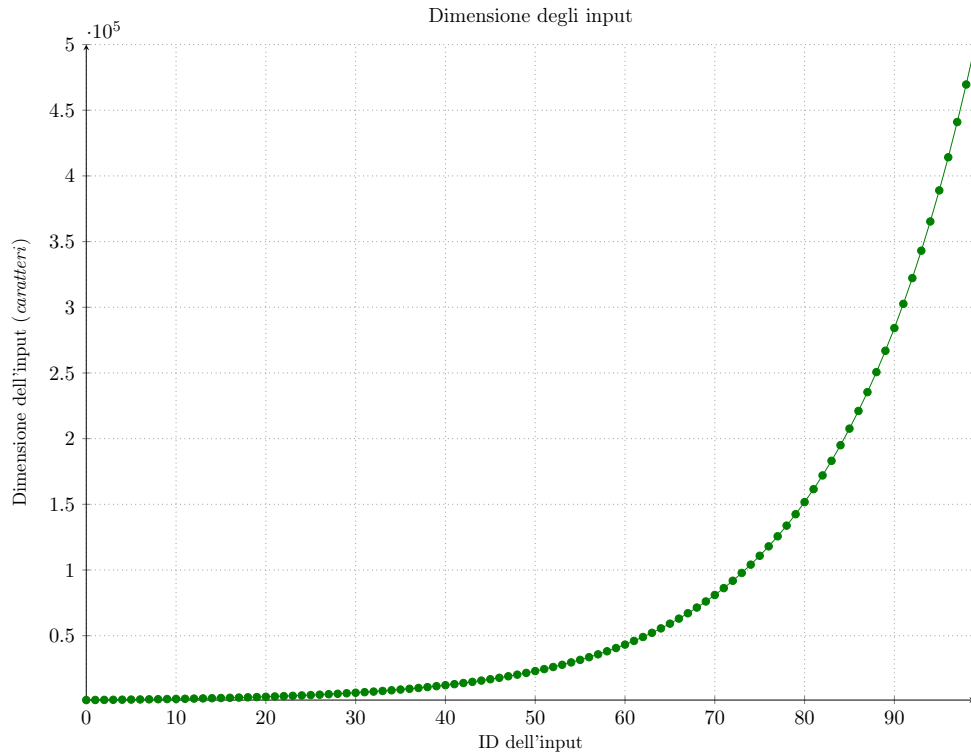
$$\begin{aligned} \lfloor A \cdot B^i \rfloor = 1000 \quad \text{se } i = 0 & \Rightarrow \lfloor A \cdot 1 \rfloor = 1000 & \Rightarrow A = 1000 \\ \lfloor 1000 \cdot B^i \rfloor = 500000 \quad \text{se } i = 99 & \Rightarrow \lfloor B^{99} \rfloor = 500 & \Rightarrow B = \sqrt[99]{500} \end{aligned}$$

```
1 # Set chosen parameters
2 FIRST_SIZE = 1_000
3 LAST_SIZE = 500_000
4 RANGE = 100
```

sources/Complexity.py

```
1 # Create an array with exponential sizes
2 # in range(FIRST_SIZE, LAST_SIZE)
3 def _getSizes(first_size, last_size, span):
4     sizes = []
5     a = first_size
6     b = (last_size/first_size)**(float(1/(span-1)))
7     for i in range(span):
8         sizes.append(int(a*(b**i)))
9     return sizes
```

sources/InputGenerator.py



4.3 Generazione dell'*array* degli input

Sono stati implementati diverse procedure per la generazione delle stringhe da utilizzare come input nel calcolo dei tempi di esecuzione degli algoritmi.

Dopo attente analisi e sperimentazioni, si è deciso di utilizzare la procedura *Random3*, descritta di seguito.

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 fino alla posizione $q - 1$, parametro scelto randomicamente nell'intervallo $[1, n]$. Ad $s(q)$ viene assegnato il carattere "d", mentre per le lettere rimanenti si procede assegnando a $s(i)$ il valore di $s((i - 1) \bmod q)$.

```

1 def InputGenerator(first_size, last_size, span):
2     sizes = _getSizes(first_size, last_size, span)
3     strings = []
4     for size in sizes:
5         strings.append(_StringGenerator_3(size))
6     return strings
7 # Create a string of size 'size' using the same
8 # method of '_InputGenerator_2', but the character in position
9 # q will be different from the others ('d')
10 def _StringGenerator_3(size):
11     q = random.randint(1, size)

```

```

12     s = (''.join(random.choices(['a', 'b', 'c'], k=q-1)))
13     s = s + 'd'
14     for i in range(q+1, size+1):
15         s = s + s[((i-1)%q)]
16     return s

```

sources/InputGenerator.py

Con questa implementazione, ci si aspetta che il risultato degli algoritmi coincida con la posizione del carattere “d”, ovvero con il parametro q .

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.

```

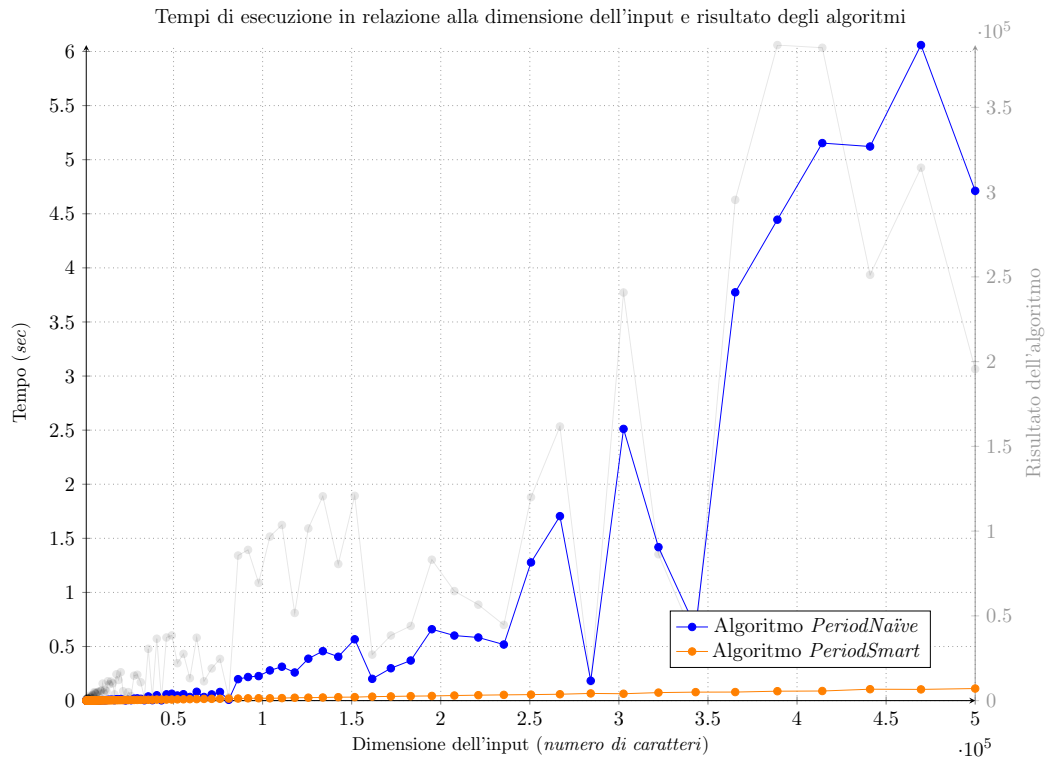
1 # Create an array with average timings from PeriodNaive
2 growth_PN = []
3 results_PN = []
4 for s in strings:
5     i = 0
6     t_passed = 0
7     while ( t_passed <= t_min ):
8         start = t.time()
9         result = PeriodNaive(s)
10        end = t.time()
11        i += 1
12        t_passed += end-start
13        results_PN.append(result)
14        growth_PN.append(t_passed/i)
15
16 # Create an array with average timings from PeriodSmart
17 growth_PS = []
18 results_PS = []
19 for s in strings:
20     i = 0
21     t_passed = 0
22     while ( t_passed <= t_min ):
23         start = t.time()
24         result = PeriodSmart(s)
25         end = t.time()
26         i += 1
27         t_passed += end-start
28         results_PS.append(result)
29         growth_PS.append(t_passed/i)

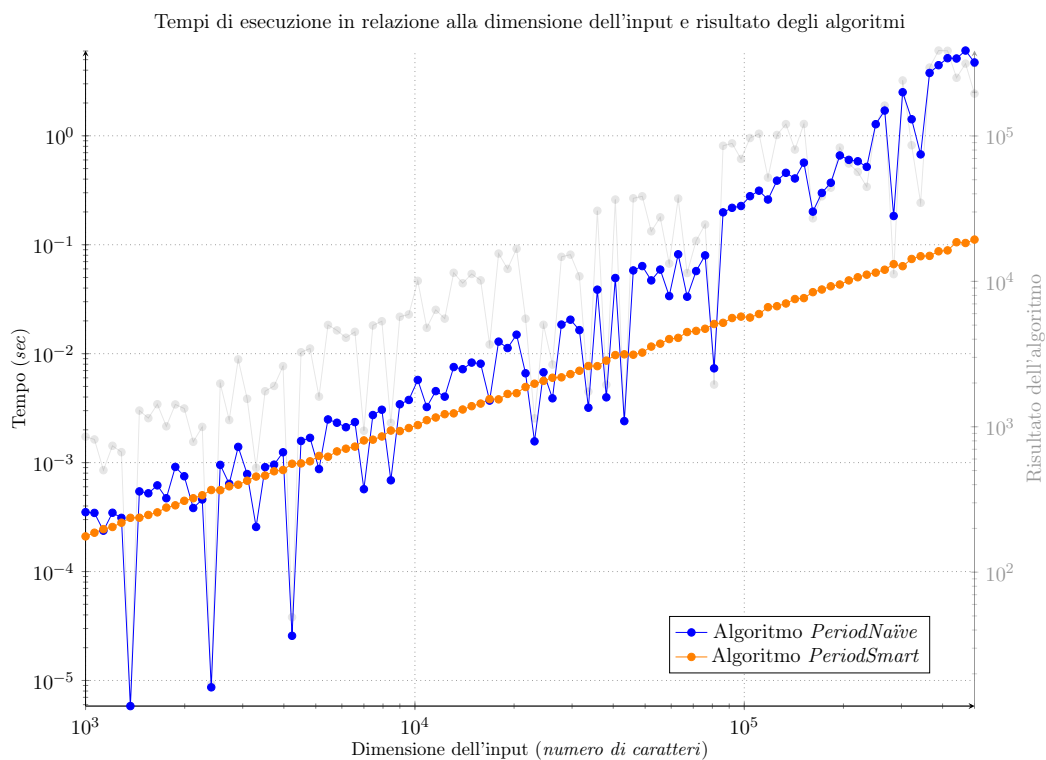
```

sources/Complexity.py

4.5 Risultati della sperimentazione

Visualizzando i risultati dei tempi di esecuzione con le rispettive lunghezze dell'input, si ottengono i grafici che seguono:





Entrambi i grafici confermano l'andamento lineare dell'algoritmo *PeriodSmart* al variare della dimensione dell'input.

Per quanto riguarda l'algoritmo *PeriodNaïve*, invece, i tempi di esecuzione sono molto più legati all'input: avendo inserito anche il risultato degli algoritmi, possiamo notare come un risultato molto inferiore alla dimensione dell'input coincida con un tempo di esecuzione molto ridotto. Al contrario, a risultati molto vicini alla dimensione dell'input coincide un tempo di esecuzione molto superiore. In generale, si può affermare dai grafici che la complessità di *PeriodNaïve* sia nell'ordine di n^2 nel caso peggiore, ovvero una complessità complessiva di $O(n^2)$.

5 Ulteriori osservazioni

5.1 Caso pessimo per l'algoritmo PeriodNaïve

5.1.1 Ideazione dell'input

Ritorniamo sul primo algoritmo proposto in questo studio

```
1 # Returns the minimum fractional period of s
2 def PeriodNaive(s):
3     assert s, "String can't be empty!"
4     n = len(s)
5     for p in range(1, n+1):
```

```

6         if(_isPeriod(s,p)):
7             return p
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        return (s[:n-p] == s[p:])

```

sources/PeriodNaive.py

e ci chiediamo come debba essere formato un input che funga da caso pessimo.

Notiamo che il codice principale è composto da un ciclo `for` con indice p varia da 1 alla lunghezza dell'input, ma termina se la porzione della stringa dall'inizio fino alla posizione p è un periodo frazionario minimo. Dunque, per massimizzare i tempi di esecuzione, dobbiamo massimizzare la posizione p che rende vero il controllo citato.

Una possibile implementazione del codice per la creazione di tale input può essere formalizzato come segue: la stringa s di lunghezza n viene generata su un alfabeto ternario $\{a,b,c\}$ in modo tale che per ogni sottostringa t che va da un carattere a non preceduto da altro a , al primo carattere c non seguito da un altro c è generata seguendo lo schema

$$t(i) = i * a + i * b + i * c$$

dove $i * n$ indica il carattere n ripetuto i volte. La stringa s così formata, viene troncata se necessario al raggiungimento del n -esimo carattere.

Esempi Un esempio di stringa formata in questo modo è:

$$s(30) = \underbrace{abc}_{t(1)} \underbrace{aabbcc}_{t(2)} \underbrace{aaabbbccc}_{t(3)} \underbrace{aaaabbbbccccc}_{t(4)},$$

mentre un altro esempio (con l'ultima sottostringa t "non completa") è

$$s(40) = \underbrace{abc}_{t(1)} \underbrace{aabbcc}_{t(2)} \underbrace{aaabbbccc}_{t(3)} \underbrace{aaaabbbbccccc}_{t(4)} \underbrace{aaaaabbbbcccc}_{t(5)}.$$

5.2 Codice dell'algoritmo

```

1  # Create a string of size 'size' in the form
2  # 'abcaabbccaaabbbccc...'
3  # This is also the worst input pattern for
4  # PeriodNaive algorithm
5  def _StringGenerator_4(size):
6      i = 0
7      s = ''
8      while len(s) <= size:
9          s = s + 'a'*i + 'b'*i + 'c'*i
10         i += 1
11     return s[:size+1]

```

5.2.1 Risultati della sperimentazione

Effettuando la sperimentazione con la stessa procedura descritta in precedenza, ma utilizzando la procedura `_StringGenerator_4` per la generazione delle stringhe, si ottiene un grafico con andamento quadratico.

