Laboratorio di Algoritmi e Strutture Dati 2020/2021 — Prima parte

Mattia Bonaccorsi — 124610 – bonaccorsi.mattia@spes.uniud.it
Muhamed Kouate — 137359 – kouate.muhamed@spes.uniud.it
Enrico Stefanel — 137411 – stefanel.enrico@spes.uniud.it
Andriy Torchanyn — 139535 – torchanyn.andriy@spes.uniud.it

26 marzo 2021

Indice

1 Il perido frazionario

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

$$s(i) = s(i + p) \qquad \forall i = 1, \ldots, (n - p) \tag{$\star$}$$

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

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

# Returns True if p is a fractional period of s,
# False otherwhise.
def _isPeriod(s,p):
    n = len(s)
    return (s[:n-p] == s[p:])
```

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:

```python
# Returns True if p is a fractional period of s,
# False otherwhise.
def _isPeriod_2(s,p):
    n = len(s)
    for j in range(0,n-p):
        if(s[j] != s[j+p]):
            return False
    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

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

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

```
# Returns the minimum fractional period of s
def PeriodSmart(s):
    n = len(s)
    pf = [0]
    for i in range (1, n):
        j = pf[i-1]
        while ((j > 0) and (s[i] != s[j])):
            j = pf[j-1]
        if (s[i] == s[j]):
            j += 1
        pf.append(j)
    maxValue = pf[n-1]
    return n - maxValue
```

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).

```
# System time resolution
import time as t

TIME_ERROR = 0.001

start = t.time()
end = t.time()

while (start == end):
    end = t.time()

r = end - start
t_min = r*((1/TIME_ERROR)+1)
```

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

4.2 Generazione dell'*array* delle dimensioni degli input

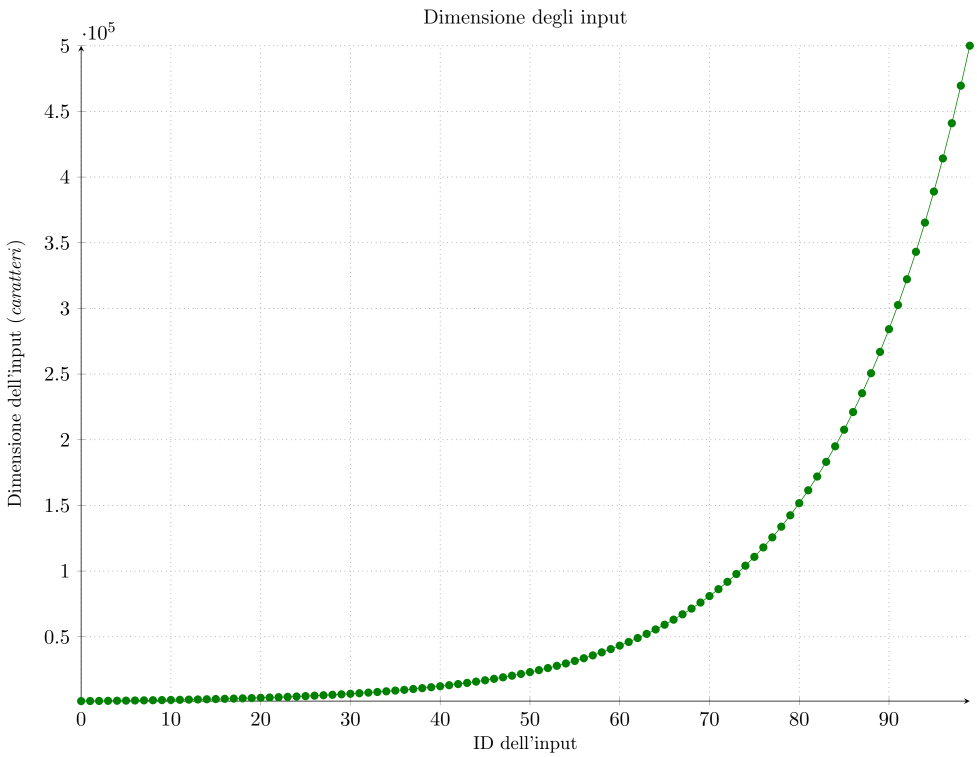
Si definisce un *array* `sizes` di lunghezze n dell'input con distribuzione esponenziale per ottenere un buon compromesso in termini di efficienza e di completezza dello studio. Tale *array* viene definito come l'insieme di tutti i valori $\lfloor A \cdot B^i \rfloor$, con $0 \leq i < 100$ e A, B calcolati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

$$\lfloor A \cdot B^i \rfloor = 1000 \quad \text{se} \quad i = 0 \quad \Rightarrow \quad \lfloor A \cdot 1 \rfloor = 1000 \quad \Rightarrow \quad A = 1000$$

$$\lfloor 1000 \cdot B^i \rfloor = 500000 \quad \text{se} \quad i = 99 \quad \Rightarrow \quad \lfloor B^{99} \rfloor = 500 \quad \Rightarrow \quad B = \sqrt[99]{500}$$

```python
# Set choosen parameters
FIRST_SIZE = 1_000
LAST_SIZE = 500_000
RANGE = 100

# Create an array with exponential sizes
# in range(FIRST_SIZE,LAST_SIZE)
sizes = []
a = FIRST_SIZE
b = (LAST_SIZE/FIRST_SIZE)**(float(1/(RANGE-1)))
for i in range(RANGE):
    sizes.append(int(a*(b**i)))
```



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 $s((i-1) \mod q)$.

```
# Set choosen method for generating inputs
METHOD = 'Random3' # 'Random1' | 'Random2' | 'Random3'

if (METHOD == 'Random1'):
    # Create an array with random strings
    # with individual lengths from sizes array
    strings = []
    for size in sizes:
        strings.append(''.join(
                random.choices(['a','b','c'], k=size)
                ))
elif(METHOD == 'Random2'):
    # Create an array with strings generated as follows:
    # generated a random position for the array,
    # then fill the string randomly until this position,
    # then copy chars from the start of the string
    # until string size is completed
    strings = []
    for size in sizes:
        q = random.randint(1, size)
        s = (''.join(random.choices(['a', 'b', 'c'], k=q)))
        for i in range(q+1, size+1):
            s = s + s[((i-1) % q)]
        strings.append(s)
elif(METHOD == 'Random3'):
    # Create an array with strings using the same
    # method of 'Random2', but the character in position
    # q will be different from the others ('d')
    strings = []
    for size in sizes:
        q = random.randint(1, size)
        s = (''.join(random.choices(['a', 'b', 'c'], k=q-1)))
        s = s + 'd'
        for i in range(q+1, size+1):
            s = s + s[((i-1) % q)]
        strings.append(s)
```

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

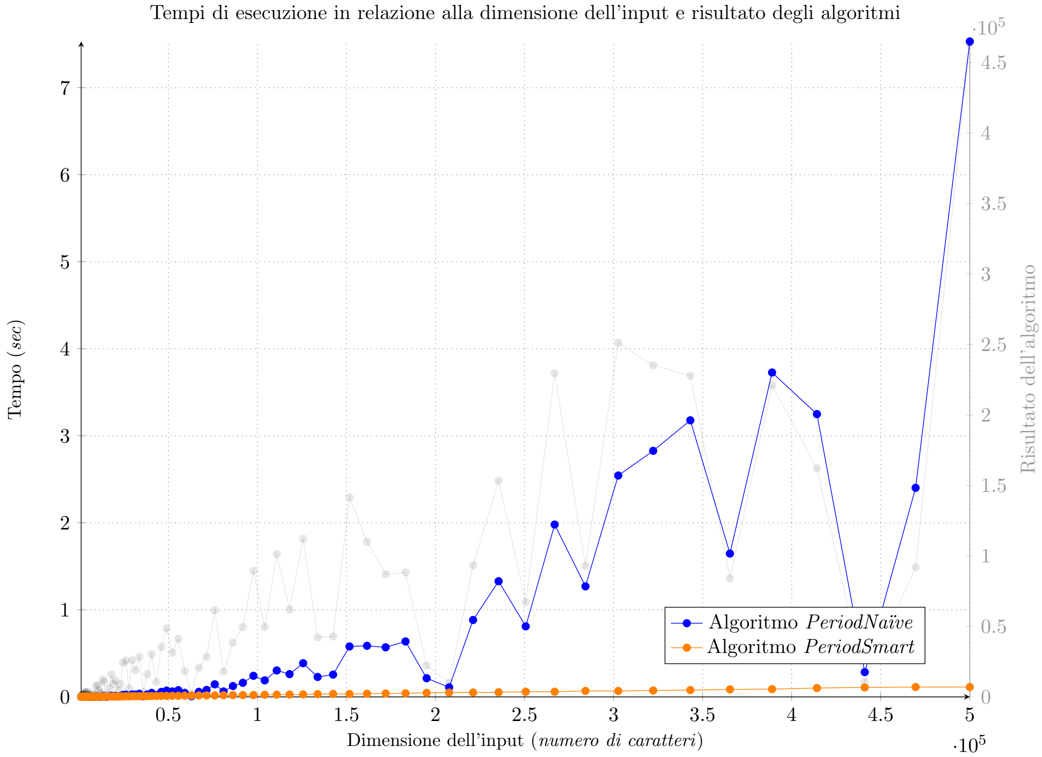
Vengono salvati in due array `growth_PN` e `growth_PS` rispettivamente i tempi di esecuzione dell'algoritmo *PeriodNaïve* e *PeriodSmart* al variare della lunghezza n della stringa fornita in input, seguendo la distribuzione indicata dall'array `strings` calcolata in precedenza.

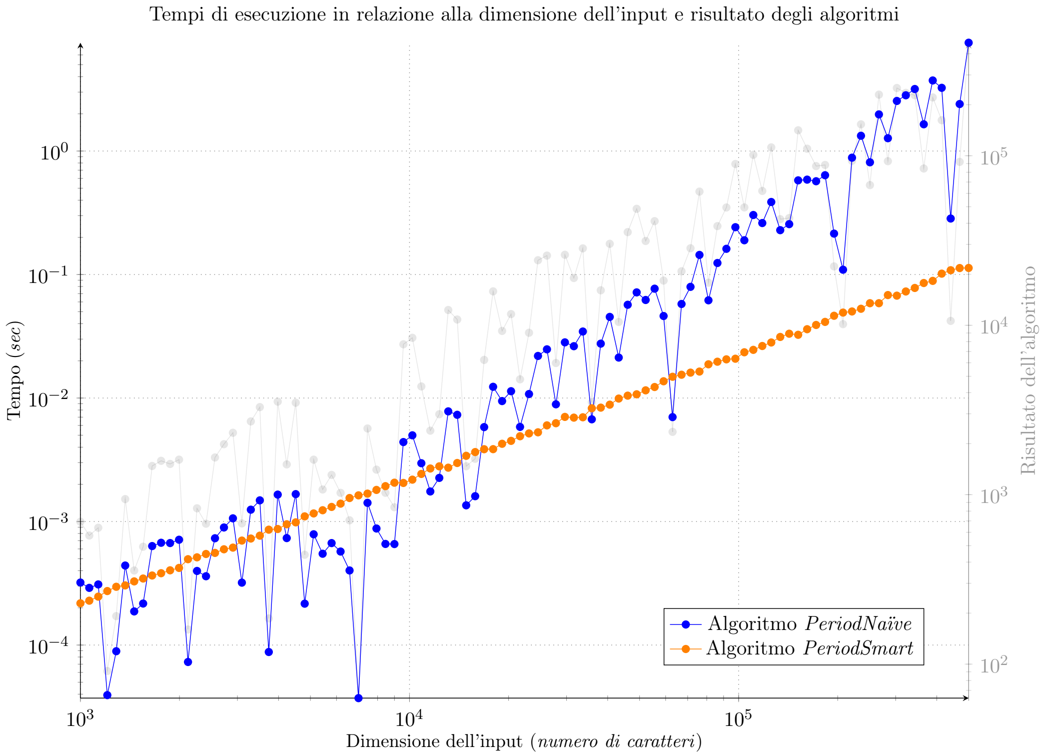
```
# Create an array with average timings from PeriodNaive
growth_PN = []
results_PN = []
for s in strings:
    i = 0
    t_passed = 0
    while ( t_passed <= t_min ):
        start = t.time()
        result = PeriodNaive(s)
        end = t.time()
        i += 1
        t_passed += end-start
    results_PN.append(result)
    growth_PN.append(t_passed/i)

# Create an array with average timings from PeriodSmart
growth_PS = []
results_PS = []
for s in strings:
    i = 0
    t_passed = 0
    while ( t_passed <= t_min ):
        start = t.time()
        result = PeriodSmart(s)
        end = t.time()
        i += 1
        t_passed += end-start
    results_PS.append(result)
    growth_PS.append(t_passed/i)
```

4.5 Risultati della sperimentazione

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


Tempi di esecuzione in relazione alla dimensione dell'input e risultato degli algoritmi
Tempo (*sec*)
Risultato dell'algoritmo
Dimensione dell'input (*numero di caratteri*)
$\cdot 10^5$
$\cdot 10^5$
Algoritmo *PeriodNaïve*
Algoritmo *PeriodSmart*


Tempi di esecuzione in relazione alla dimensione dell'input e risultato degli algoritmi
Tempo (*sec*)
Risultato dell'algoritmo
Dimensione dell'input (*numero di caratteri*)
Algoritmo *PeriodNaïve*
Algoritmo *PeriodSmart*

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 $O(n^2)$.