Relazione di Algoritmi e Strutture Dati

Studenti	Matricola	Mail
Manuel Zuttion	147404	147404@spes.uniud.it
Cristian Murtas	150521	150521@spes.uniud.it
Sebastiano Vitri	147640	147640@spes.uniud.it

Indice

Obbiettivi e implementazione

- Implementazione degli algoritmi *PeriodNaive* e *PeriodSmart* ;
- Implementazione dei 4 metodi per la generazione delle stringhe;
- Scrittura di un algoritmo per la misurazione dei tempi di esecuzione degli algoritmi applicati a tutti i tipi di stringhe;
- Misurazione e calcolo della varianza sui tempi di esecuzione (facoltativo);
- Calcolo della distribuzione di probabilità del periodo per i 4 metodi per la generazione delle stringhe.

Definizione di periodo per una stringa

Data una stringa S composta da *n* caratteri, il periodo è un numero k tale che S[i] = S[i+k] per ogni i $\leq$ n-k.
N.B. con indicizzazione dei caratteri della stringa a partire da 1.

La sottostringa di caratteri di lunghezza *k* può essere ripetuta un numero non intero di volte in S, ovvero *n* non è necessariamente un multiplo di *k*.

Spiegazione dell'algoritmo *PeriodNaive*

L'algoritmo utilizza un ciclo *for* con un indice *i* che varia da 1 a *n* e alla prima iterazione che soddisfa la seguente proprietà:
$\forall$i = 1, . . . , n - p s(i) = s(i+p)
La proprietà sopra elencata può essere verifica in due modi:
1. Usando un ciclo *for* secondario e controllando che s(i) = s(i + p) $\forall$i = 1, . . . , n - p;
2. Alternativamente è possibile utilizzare le funzioni *strcmp* e *strsub* del linguaggio *C* per confrontare che il prefisso s[1, n-p] sia congruente con il suffisso s[p+1, n].

La nostra implementazione ha usato il secondo metodo di verifica.

Spiegazione dell'algoritmo *PeriodSmart*

Abbiamo bisogno di definire il **bordo** di una stringa come:
Una qualunque stringa t che sia contemporaneamente prefisso e suffiso proprio di s.
Si osservi che *p* è un periodo frazionario di S se e soltanto se *p* = |S| - *r* dove *r* è la lunghezza del bordo massimo di S.
Di conseguenza, il nostro problema originale si è ridotto al calcolo del bordo massimo per una stringa S.

Generazione delle stringhe

Per il calcolo della lunghezza delle stringhe non è stata impiegata *b* come costante moltiplicativa a ogni iterazione.
Abbiamo, invece, preferito introdurre la sua formula durante il calcolo della lunghezza della stringa.
Ciò ha permesso di evitare la propagazione dell'errore dovuto alla moltiplicazione iterativa di valori affetti da approssimazioni floating-point.

Abbiamo utilizzato un valore di *a* pari a 1000, la quale risulta essere la lunghezza della stringa S alla prima iterazione, e pow(...), che matematicamente equivale a *b^i*. In questo modo calcoliamo la lunghezza della stringa senza propagazione dell'errore.
N.B. Per la generazione delle stringhe abbiamo utilizzato un alfabeto basato su 2 caratteri {a, b}

Metodi per la generazione delle stringhe

1. Il primo metodo prevede la generazione degli *n* caratteri in modo pseudo-casuale.
2. Si ha la presenza di un parametro *q* generato pseudo casualmente tra 1 e *n*, la stringa verrà quindi generata in modo pseudo-casuale nell'intervallo *[1, q]*.
   Successivamente tale sottostringa verrà ripetuta fino al termine della stringa S.
   Con l'operazione modulo si evita un indexing out of bound.
3. È una variante del secondo in cui è previsto che nella stringa S in posizione *S[q]* venga inserito un carattere speciale differente da tutti quelli generati.
4. La quarta variante prevede la costruzione deterministica di una stringa di modo che possa fungere da caso pessimo nell'analisi di complessità.

Misurazione dei tempi di esecuzione dei due algoritmi

Per la parte di misurazione dei tempi abbiamo impiegato i metodi della libreria di *C++*.
È stata poi calcolata la risoluzione del clock di sistema, utile al fine di calcolare l'intervallo minimo di tempo misurabile:
$Tmin = R(\frac{1}{e} + 1)$
Il valore minimo di tempo misurabile è stato adoperato per la misurazione dei tempi degli algoritmi sulle stringhe aventi una lunghezza contenuta. Per fare ciò ci siamo serviti di un ciclo while, il quale itera l'algoritmo sulla stessa stringa finché il tempo cumulativo dell'esecuzione non risulti maggiore o uguale a *Tmin* .
Dopodiché il valore di tempo ottenuto viene diviso per il numero di iterazioni eseguite, tenute in memoria da un contatore.

Calcolo della varianza

Questa parte del progetto riprende parzialmente la logica di funzionamento del codice per la misurazione dei tempi. Siamo partiti con la creazione di un vettore contenente tutte le lunghezze delle stringhe nell'intervallo *[1000, 500000]*.
Successivamente abbiamo iterato 20 volte la misurazione dei tempi di esecuzione per ogni *n*, variando la stringa ad ogni iterazione.
Tutte le misurazioni eseguite, relative alla specifica lunghezza, sono state poi scritte in un file *.csv* per l'analisi eseguita con RStudio.

Calcolo delle distribuzioni di probabilità

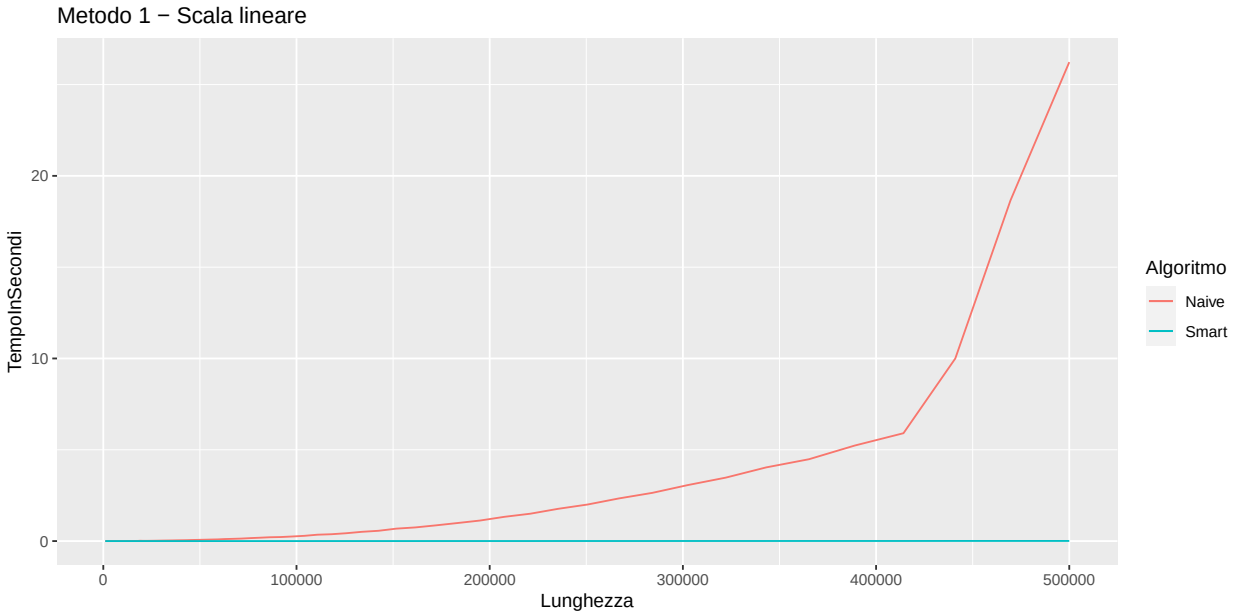
Per il calcolo delle distribuzioni si replica per 500 volte la generazione di stringhe di lunghezza 100 per ognuno dei 4 metodi.
Per ciascuna iterazione si calcola il periodo della stringa generata, incrementando di 1 il valore nella posizione corrispondente al periodo nell'array delle occorrenze.

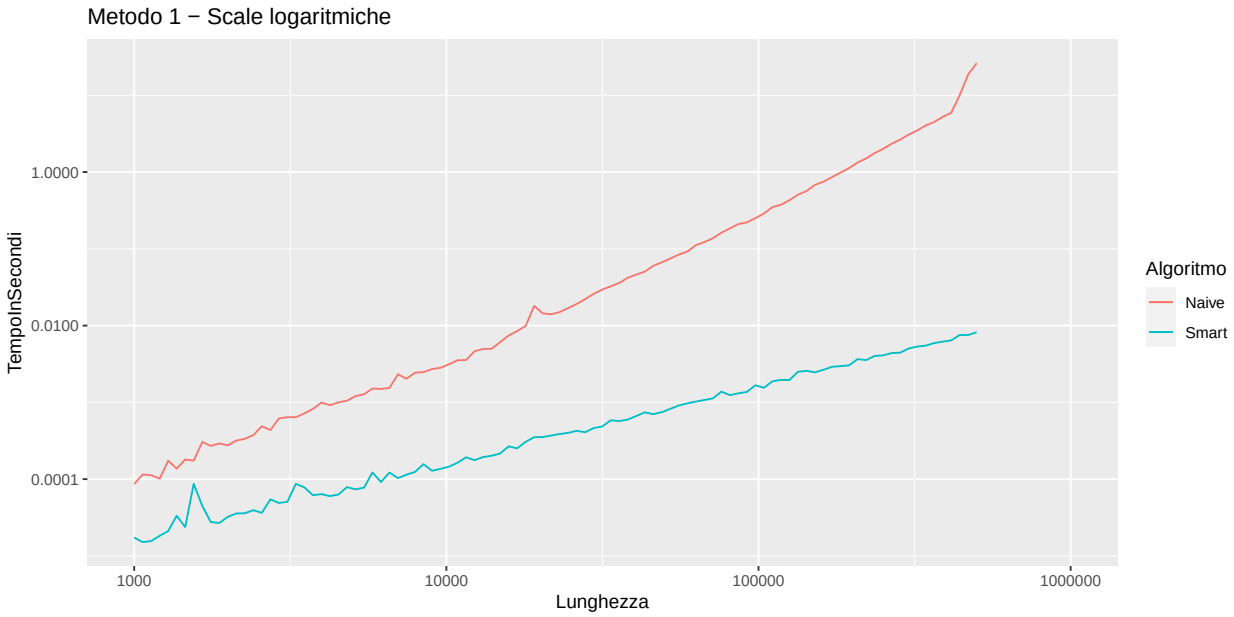
I dati memorizzati nell'array delle occorrenze vengono poi trascritti in un file *.csv* per l'analisi tramite RStudio.

Presentazione risultati

Grafici di confronto tra i tempi di esecuzione di *PeriodNaive* e *PeriodSmart* con scala lineare e doppiamente logaritmica.

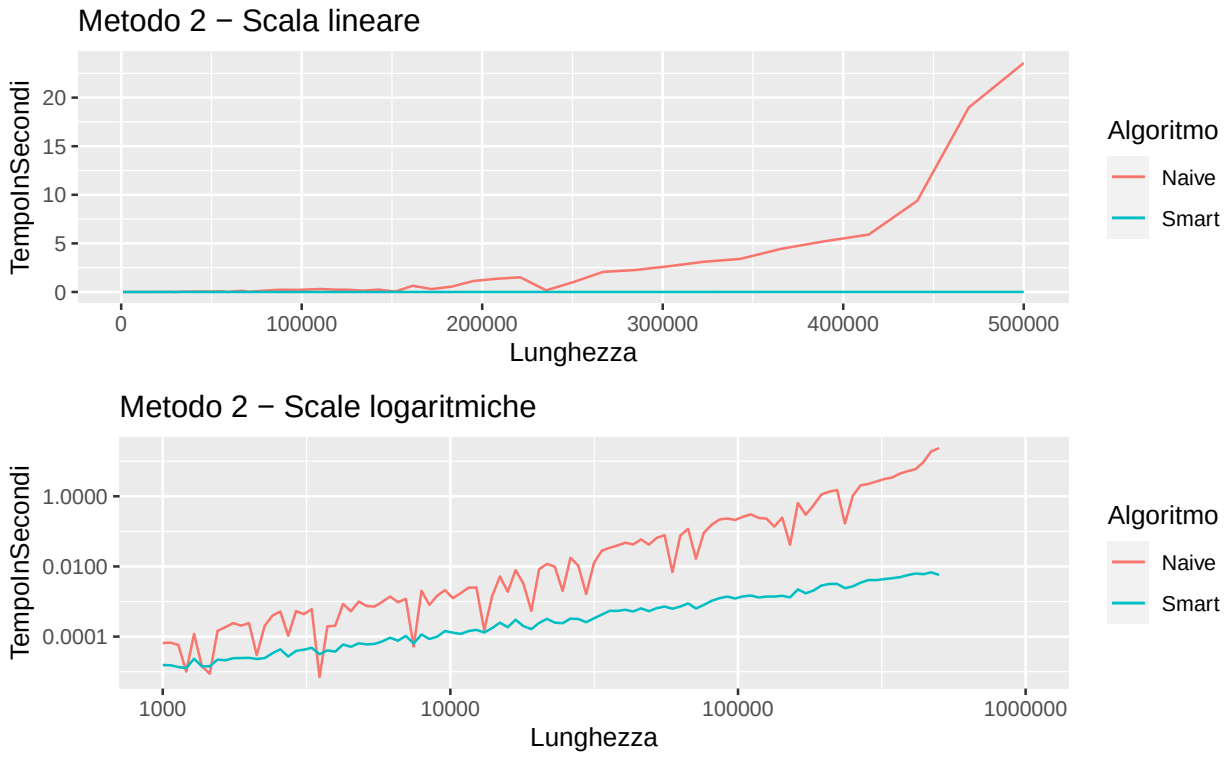
Dall'analisi di complessità nel caso medio dei due algoritmi si è ottenuto un andamento quadratico per l'algoritmo *PeriodNaive*.
Per quanto riguarda l'algoritmo *PeriodSmart* la complessità è risultata lineare.



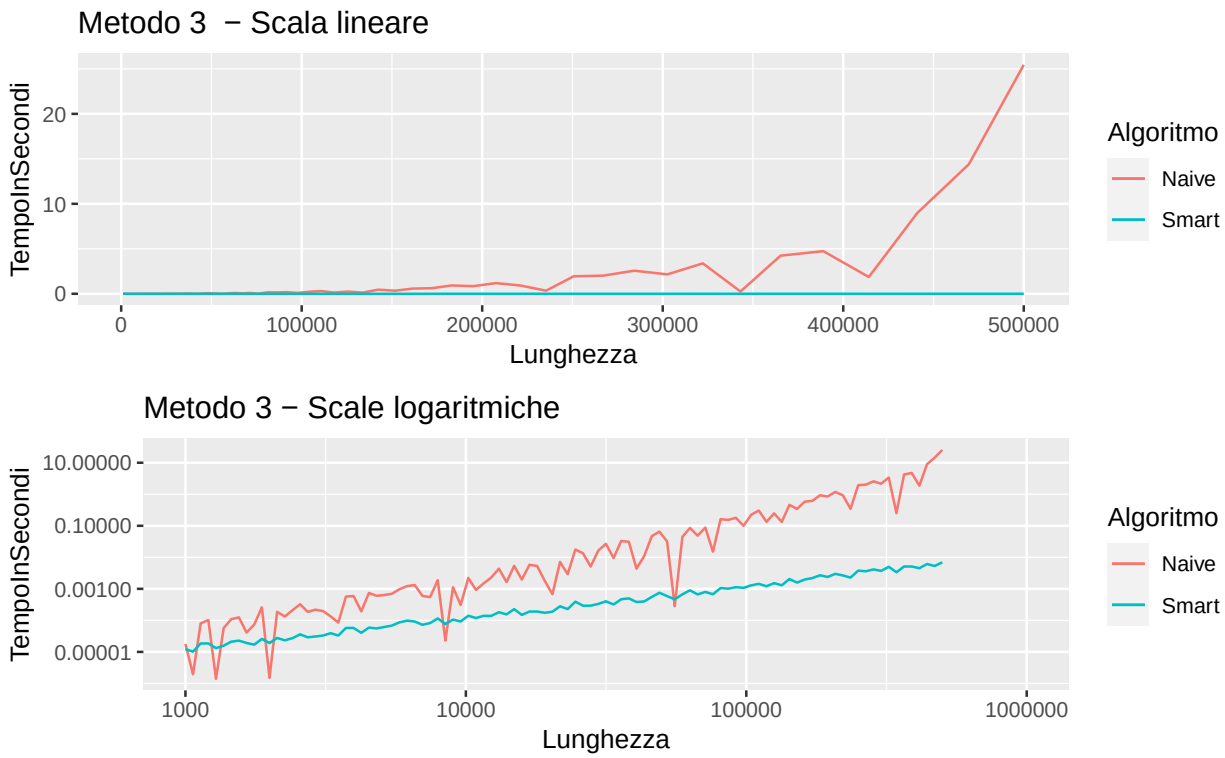


Nel primo grafico si osserva come l'algoritmo *PeriodSmart* risulti essere molto più performante rispetto al suo rivale.
Nel secondo grafico, invece, la curva dell'algoritmo *PeriodNaive* ha pendenza doppia rispetto alla curva dell'algoritmo *PeriodSmart*.

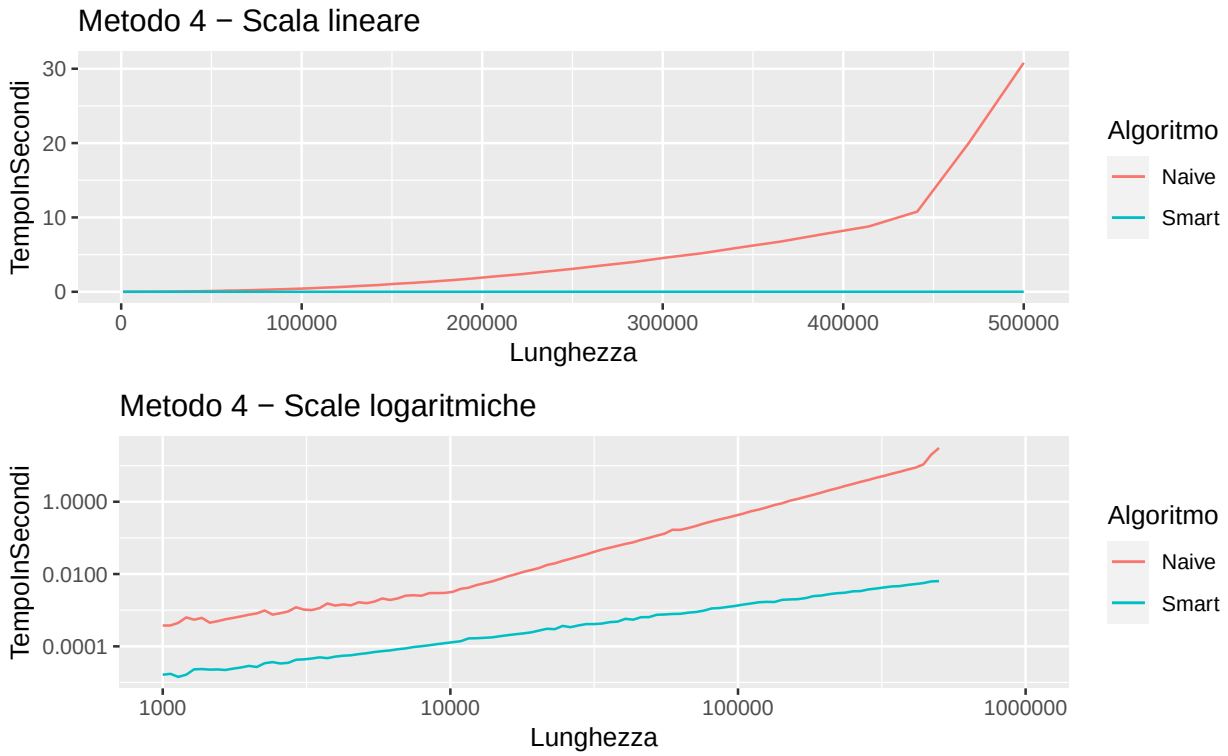
Esso è indice di un andamento rispettivamente quadratico e lineare, come evidenziato dall'analisi di complessità svolta a lezione.



In questo contesto il metodo di generazione prevede la scelta del periodo in modo pseudo-casuale, rendendo le curve molto frastagliate, soprattutto per l'algoritmo *PeriodNaive*. Tale andamento mette in evidenza come *PeriodNaive* termini anticipatamente l'esecuzione nel caso di un periodo più contenuto della stringa.
Tuttavia, accade in maniera meno evidente nel caso dell'algoritmo *PeriodSmart*.

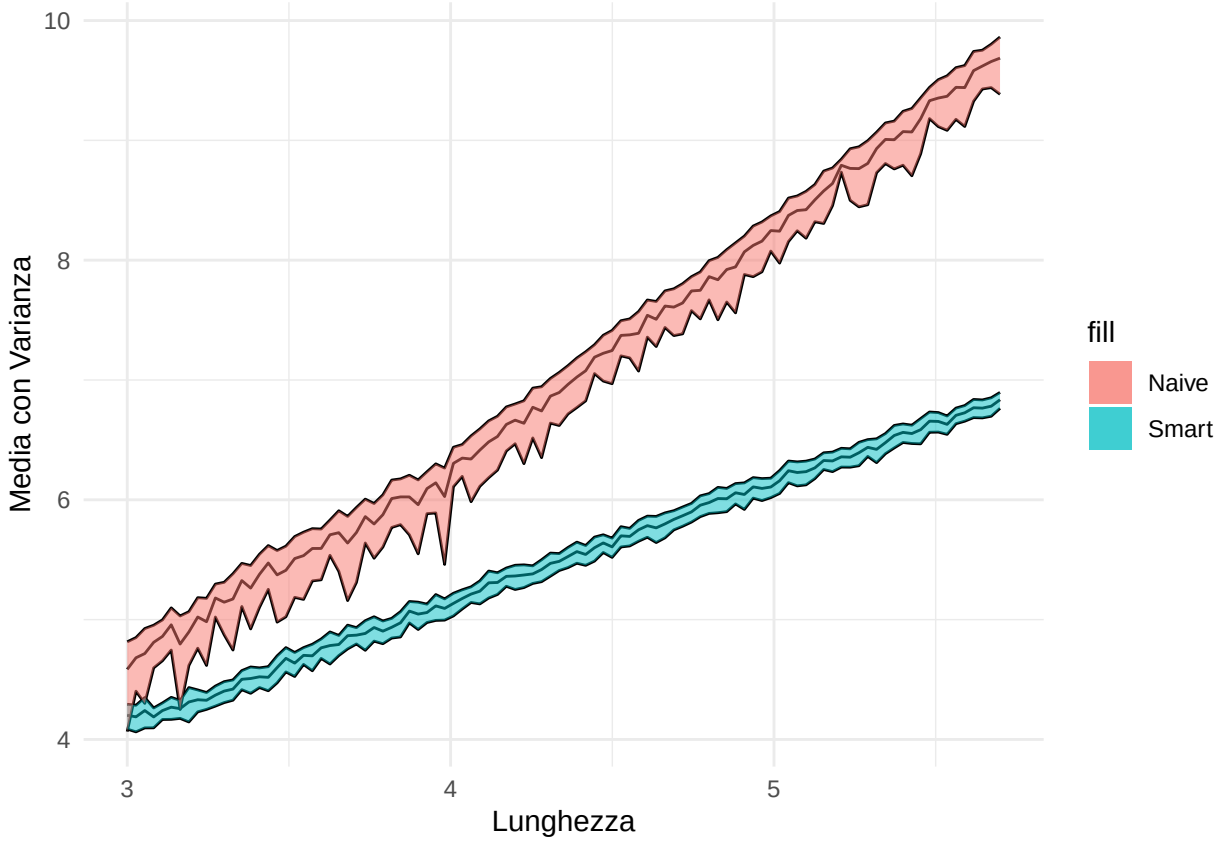


Rispetto al metodo 2, si prevede che l'ultimo carattere componente il periodo sia diverso da tutti quelli generati anteriormente.
Dai grafici non si evince nessuna sostanziale differenza nell'andamento della curva.



Come precedentemente discusso, se si assume il caso pessimo, ovvero il caso in cui il periodo è sempre pari alla lunghezza della stringa, si noterà che la curva relativa all'algoritmo *PeriodNaive* risulta essere meno frastagliata e più fedele alle aspettative. Allo stesso tempo, osservando la controparte *PeriodSmart*, si mette in evidenza come esso non risenta del caso pessimo ottenendo dei valori in linea con quelli ricavati tramite i metodi sopra descritti.

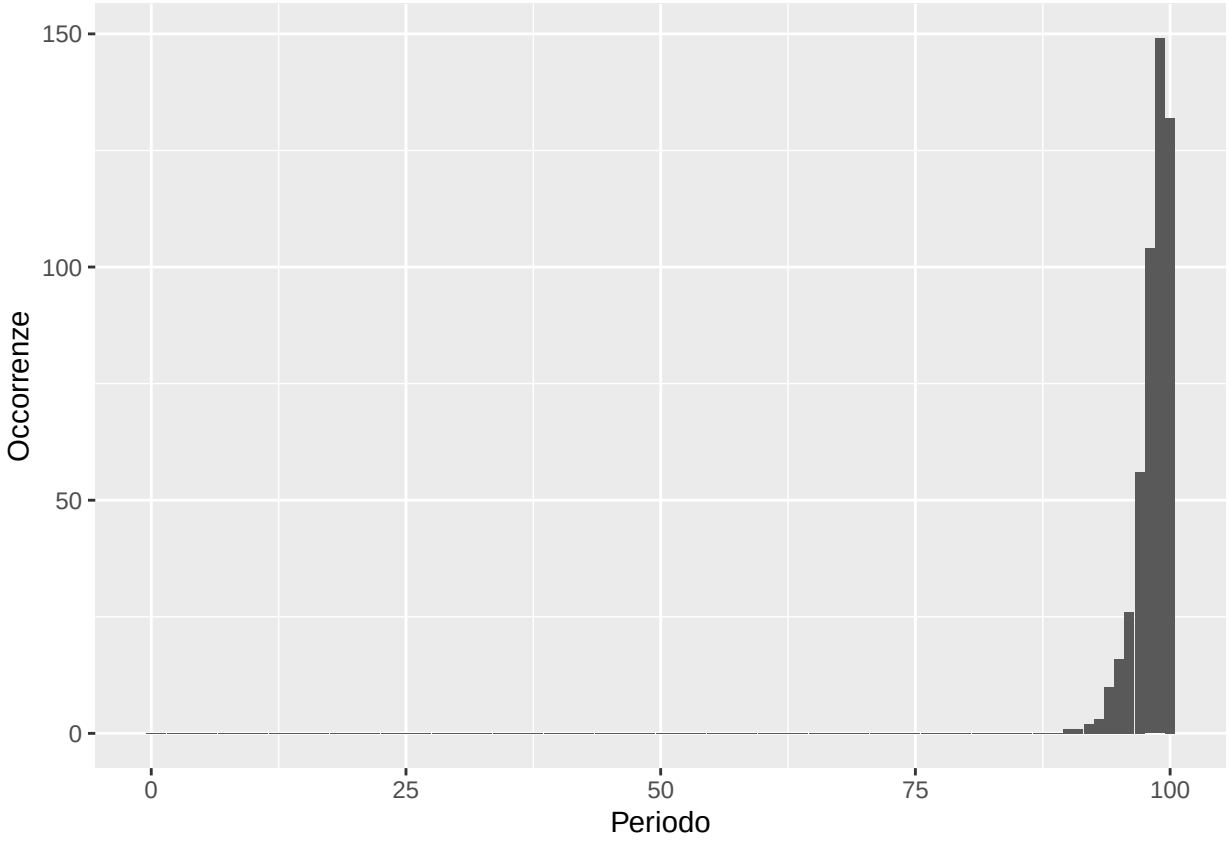
Presentazione dei dati ottenuti per il calcolo della varianza



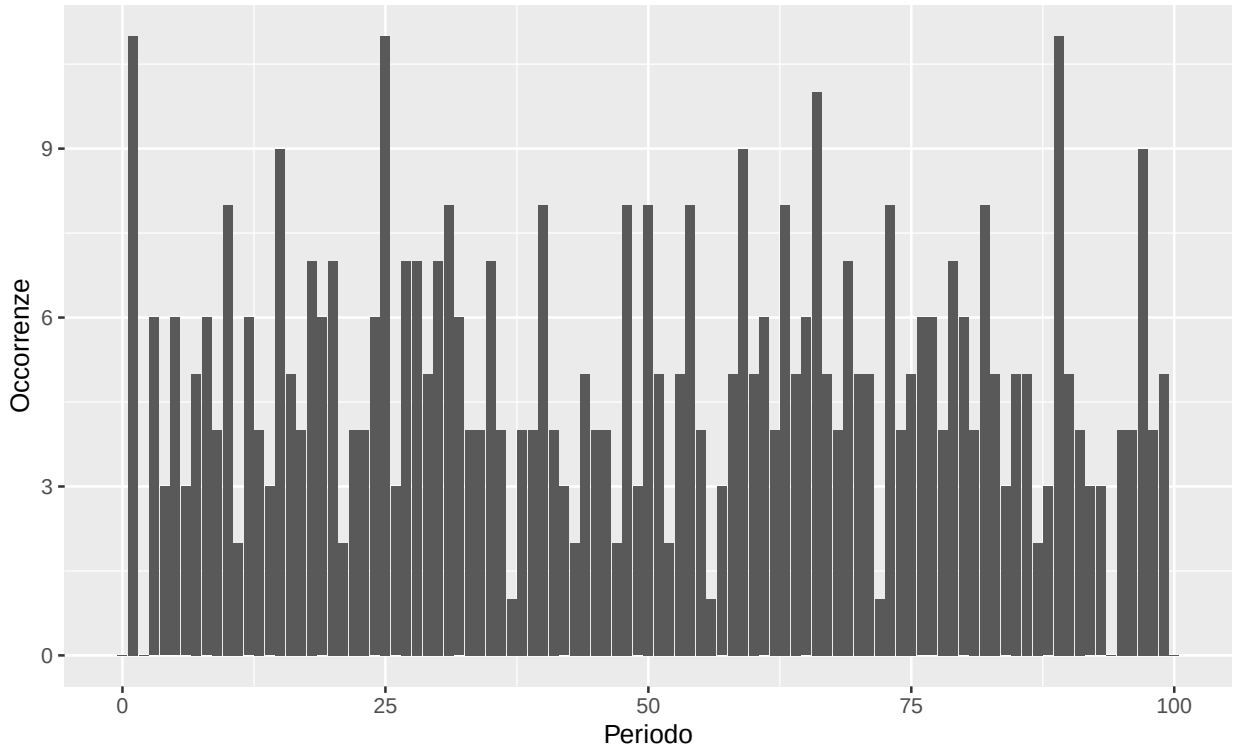
Dal grafico si rileva che l'algoritmo *PeriodSmart* sia caratterizzato da una varianza più contenuta.
La causa da noi ipotizzata sarebbe che la complessità dell'algoritmo sopracitato non sia influenzata dalla lunghezza del periodo.
D'altro canto l'algoritmo *PeriodNaive* mostra una varianza maggiore, giacché i tempi di esecuzione per una data stringa s dipendono, in maggior misura, dalla lunghezza del periodo.

Distribuzioni di probabilità del periodo

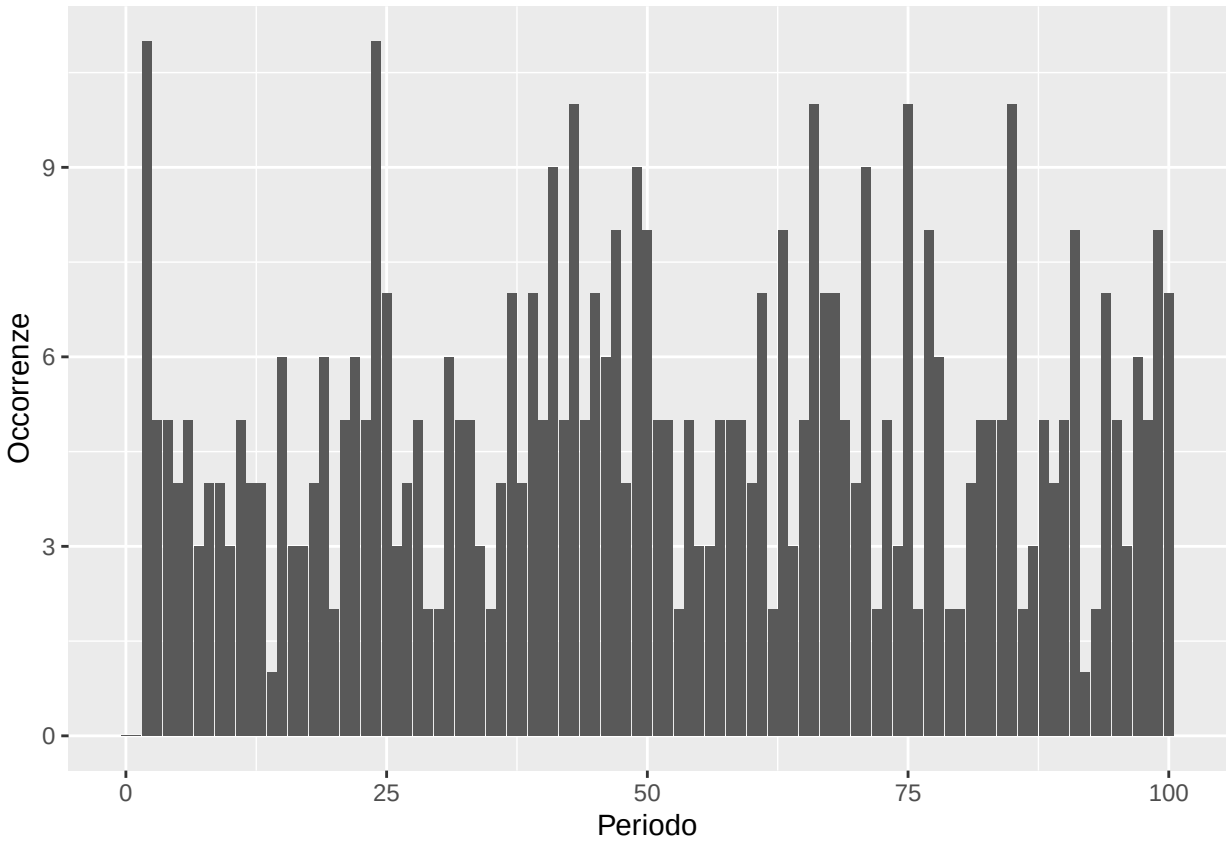
con i 4 metodi di generazione



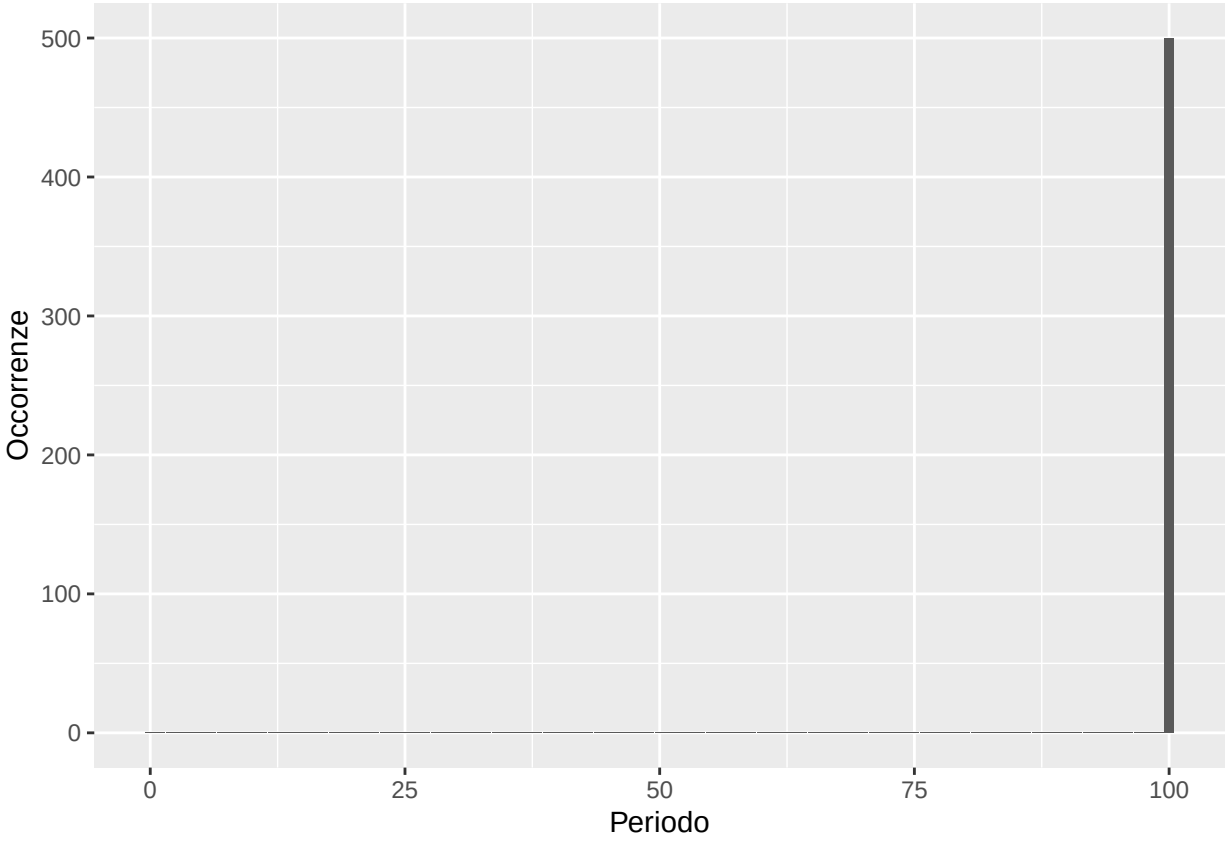
Questo metodo prevede la generazione della stringa con tutti i caratteri disponibili nell'alfabeto in maniera pseudo-casuale.
Come si deduce dal grafico, il periodo tende ad essere prossimo alla lunghezza della stringa. Avendo utilizzato un alfabeto composto da due caratteri (a, b) la probabilità che una determinata sequenza si ripeta è molto bassa, di conseguenza il periodo tende ad essere molto alto.



Il secondo metodo implica la generazione randomica di un indice intero *q*, utilizzato per generare la sottostringa da *[1, q].* In seguito si procederà ripetendo tale sottostringa fino a che la lunghezza *n* sarà soddisfatta.
La distribuzione risulta essere ben distribuita su tutto l'intervallo, poiché la scelta di *q* ha una probabilità di $\frac{1}{n}$.
Nel nostro caso *n* è 100.



Nel grafico si illustra la variante del secondo metodo.
Rispetto a quest'ultimo la distribuzione delle occorrenze risulta essere molto più uniforme.



Analisi del quarto metodo di generazione

Prevede la costruzione deterministica di una stringa che possa fungere da caso pessimo.
In maniera evidente si può notare che tutte le stringhe generate hanno come periodo la lunghezza n (100).

Considerazioni finali

Dai dati ottenuti abbiamo potuto confermare empiricamente la complessità degli algoritmi analizzati.
Il quarto metodo di generazione delle stringhe è stato il più efficace a evidenziare la discrepanza di complessità tra i due algoritmi, specialmente nel grafico logaritmico.
I grafici ricavati relativi alla varianza suggeriscono come i due algoritmi risentano in maniera differente della lunghezza del periodo.
La varianza maggiore osservata in *PeriodNaive* fa supporre che i suoi tempi di esecuzione, per una data stringa s, dipendano fortemente dal periodo piuttosto che dalla lunghezza della stringa in sé.