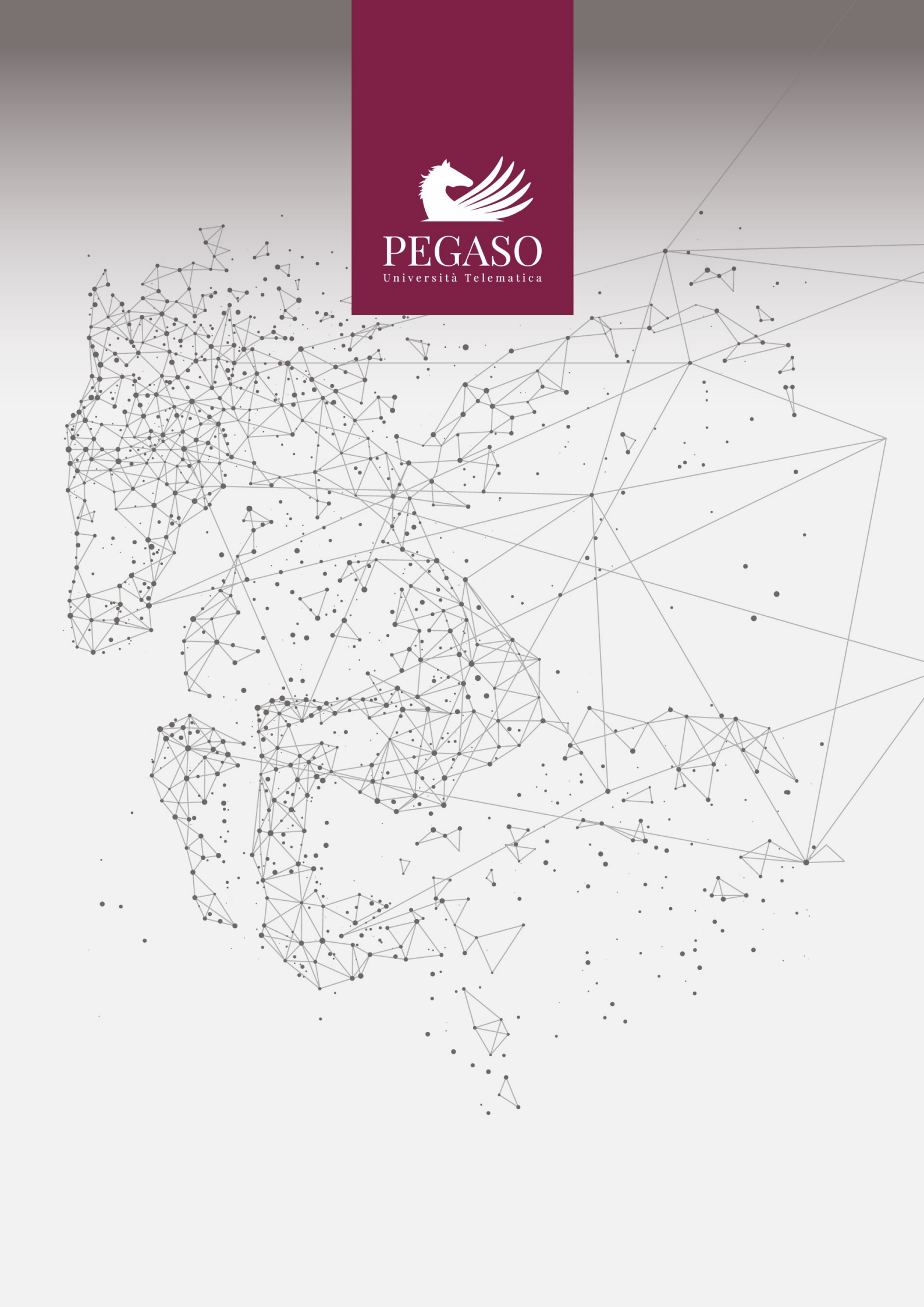
PEGASO
Università Telematica

Indice

1. Introduzione

Nel campo della programmazione, esistono numerose strategie utilizzate per risolvere problemi e ottenere risultati in modo efficiente. Tra queste strategie, si distinguono gli **algoritmi deterministici** e gli **algoritmi probabilistici**.

Gli algoritmi deterministici sono basati su una serie di passi logici che producono una soluzione esatta e deterministica al problema in questione. Questi algoritmi sono utilizzati quando la soluzione esatta del problema può essere ottenuta con certezza e senza ambiguità. Gli algoritmi deterministici sono spesso utilizzati per risolvere problemi di ottimizzazione, ricerca e ordinamento.

D'altra parte, gli algoritmi probabilistici producono soluzioni approssimate al problema in questione, basandosi sulla probabilità invece che sulla certezza. Questi algoritmi sono utilizzati quando la soluzione esatta del problema è difficile o impossibile da ottenere con certezza, ma una soluzione approssimata può essere sufficiente per le esigenze dell'applicazione. Gli algoritmi probabilistici sono spesso utilizzati per la simulazione di fenomeni fisici, la ricerca di informazioni su Internet e la criptografia.

Le tecniche di programmazione si basano sulla scelta dell'algoritmo più appropriato per risolvere un determinato problema. La scelta tra algoritmi deterministici e probabilistici dipende dalle specifiche esigenze dell'applicazione e dalle caratteristiche del problema da risolvere. In generale, gli algoritmi deterministici sono preferiti quando la soluzione esatta del problema può essere ottenuta con certezza e senza ambiguità, mentre gli algoritmi probabilistici sono preferiti quando una soluzione approssimata può essere sufficiente.

Nella scelta tra algoritmi deterministici e probabilistici, è importante considerare il tempo di esecuzione richiesto, la precisione della soluzione richiesta e le caratteristiche specifiche del problema da risolvere. In alcuni casi, può essere opportuno utilizzare una combinazione di algoritmi deterministici e probabilistici per ottenere la soluzione migliore possibile.

Gli algoritmi probabilistici sono algoritmi che utilizzano la probabilità come base per prendere decisioni. Questo tipo di algoritmo è molto utilizzato in diversi campi, come la teoria dei giochi, la crittografia, l'intelligenza artificiale e molti altri.

In generale, un algoritmo probabilistico è composto da due parti:

- la fase di preparazione
- la fase di esecuzione

Nella fase di preparazione, l'algoritmo prepara i dati necessari per l'esecuzione dell'algoritmo stesso; nella fase di esecuzione, l'algoritmo utilizza i dati preparati nella fase precedente per prendere una decisione basata sulla probabilità.

Uno degli algoritmi probabilistici più conosciuti è l'algoritmo di **Monte Carlo**. Questo algoritmo utilizza una serie di numeri casuali per simulare il comportamento di un sistema complesso. L'algoritmo di Monte Carlo è utilizzato in molti campi, tra cui la finanza, l'ingegneria, la fisica e molti altri.

Un altro algoritmo probabilistico noto è l'algoritmo di **Las Vegas**. Questo algoritmo è simile all'algoritmo di Monte Carlo, ma invece di utilizzare numeri casuali per simulare il comportamento di un sistema complesso, utilizza l'esecuzione di un algoritmo deterministico per trovare la soluzione corretta. L'algoritmo di Las Vegas è spesso utilizzato in applicazioni crittografiche e di sicurezza informatica.

In generale, gli algoritmi probabilistici sono molto utili perché possono essere molto efficienti e possono risolvere problemi che altrimenti sarebbero troppo complessi da risolvere con algoritmi deterministici. Tuttavia, essi richiedono una comprensione approfondita della probabilità e della teoria della probabilità per essere utilizzati in modo efficace.

2. Algoritmo di Montecarlo

L'algoritmo di Monte Carlo è un algoritmo probabilistico che utilizza numeri casuali per simulare il comportamento di un sistema. Questo algoritmo è ampiamente utilizzato in molti campi, tra cui la finanza, l'ingegneria, la fisica, la biologia, la chimica e molti altri.

Il funzionamento dell'algoritmo di Monte Carlo si basa sulla generazione di numeri casuali che vengono utilizzati per simulare il comportamento del sistema. Questi numeri casuali vengono generati da un **generatore di numeri casuali**, che produce sequenze di numeri che soddisfano determinate proprietà statistiche.

Una volta generati i numeri casuali, l'algoritmo di Monte Carlo utilizza questi numeri per stimare una quantità di interesse.

Ad esempio, supponiamo di dover stimare l'area di una figura complessa. Invece di calcolare l'area esatta, l'algoritmo di Monte Carlo genera numeri casuali che rappresentano i punti all'interno della figura e conta quanti di questi cadono all'interno della figura stessa. Utilizzando la proporzione tra i punti all'interno della figura e il numero totale di punti generati, l'algoritmo di Monte Carlo stima l'area della figura.

Supponiamo di dover calcolare l'area di un cerchio di raggio 1; l'idea è quella di generare un grande numero di punti casuali all'interno di un quadrato di lato 2, centrato nell'origine, e contare quanti di questi punti cadono all'interno del cerchio unitario:

```
import random

# Numero di punti generati
num_points = 1000000

# Contatori per i punti all'interno e all'esterno del cerchio
points_inside_circle = 0
points_outside_circle = 0

# Generazione dei punti casuali all'interno del quadrato
for i in range(num_points):
    # Coordinata x del punto
    x = random.uniform(-1, 1)
    # Coordinata y del punto
    y = random.uniform(-1, 1)

    # Calcolo della distanza del punto dall'origine
    distance = x**2 + y**2

    # Se il punto si trova all'interno del cerchio unitario
```

```
    if distance <= 1:
        points_inside_circle += 1
    else:
        points_outside_circle += 1

# Calcolo dell'area del cerchio
circle_area = 4 * (points_inside_circle / num_points)

print("Area del cerchio di raggio 1: ", circle_area)
```

In questo esempio, la variabile num_points rappresenta il numero di punti casuali generati, mentre i contatori points_inside_circle e points_outside_circle contano rispettivamente i punti all'interno e all'esterno del cerchio unitario.

Alla fine dell'iterazione, l'area del cerchio viene stimata dividendo il numero di punti all'interno del cerchio per il numero totale di punti generati e moltiplicando il risultato per 4 (perché il quadrato di lato 2 ha area 4).

In generale, il funzionamento dell'algoritmo di Monte Carlo può essere suddiviso in tre fasi principali: generazione dei numeri casuali, simulazione del sistema e calcolo della soluzione approssimata.

1. La prima fase consiste nella **generazione di una serie di numeri casuali**. Questi numeri casuali sono generati da un generatore di numeri casuali, che produce sequenze di numeri che soddisfano determinate proprietà statistiche. Questi numeri casuali sono utilizzati per simulare il comportamento del sistema.
2. Nella seconda fase, l'algoritmo utilizza i numeri casuali generati nella fase precedente per **simulare il comportamento del sistema**. Ad esempio, se si vuole simulare il comportamento di una particella in un campo magnetico, si possono generare numeri casuali per rappresentare la posizione e la velocità della particella in ogni istante di tempo. Utilizzando questi numeri casuali, l'algoritmo può simulare il comportamento della particella nel campo magnetico.
3. Nella terza fase, l'algoritmo utilizza i dati raccolti nella fase di simulazione per **calcolare una soluzione approssimata al problema**. Ad esempio, se si vuole calcolare l'area di una figura complessa, l'algoritmo può generare numeri casuali per rappresentare i punti all'interno della figura e utilizzare la proporzione tra i punti all'interno della figura e il numero totale di punti generati per stimare l'area della figura.

Come si intuisce, la principale limitazione dell'algoritmo di Monte Carlo è che richiede la generazione di un grande numero di numeri casuali per ottenere una stima accurata; inoltre, l'algoritmo può essere influenzato dalla scelta del generatore di numeri casuali utilizzato.

L'algoritmo di Monte Carlo ha molte applicazioni concrete in diversi campi:

- **Finanza**: l'algoritmo è ampiamente utilizzato nel settore finanziario per valutare le opzioni e i derivati; ad esempio, per valutare un'opzione di acquisto su un'azione, si utilizza l'algoritmo per simulare il comportamento del prezzo dell'azione nel tempo e valutare l'opzione in base ai risultati della simulazione.
- **Fisica**: l'algoritmo è utilizzato in fisica per simulare il comportamento dei sistemi fisici; ad esempio, per simulare il comportamento di un sistema di particelle, si utilizza l'algoritmo per generare numeri casuali che rappresentano la posizione e la velocità delle particelle in ogni istante di tempo.
- **Biologia**: l'algoritmo viene utilizzato in biologia per simulare il comportamento dei sistemi biologici; ad esempio, per simulare il comportamento di una proteina, si utilizza l'algoritmo per generare numeri casuali che rappresentano la posizione e l'orientamento della proteina in ogni istante di tempo.
- **Chimica**: l'algoritmo viene utilizzato in chimica per simulare il comportamento dei sistemi chimici; ad esempio, per simulare il comportamento di una reazione chimica, si utilizza l'algoritmo per generare numeri casuali che rappresentano la posizione e la velocità delle particelle chimiche in ogni istante di tempo.
- **Ingegneria**: l'algoritmo viene utilizzato in ingegneria per valutare la sicurezza e l'affidabilità dei sistemi; ad esempio, per valutare la resistenza di un ponte, si utilizza l'algoritmo per simulare il comportamento delle forze che agiscono sul ponte in diverse condizioni.

Focalizzandoci ad es. nel settore finanziario, immaginiamo di voler valutare prodotti finanziari complessi, come le opzioni, i derivati e altri strumenti finanziari: questi prodotti finanziari dipendono dal prezzo del sottostante, ad esempio il prezzo di un'azione o di un'obbligazione.

Il prezzo del sottostante può fluttuare nel tempo in modo imprevedibile, rendendo difficile calcolare il prezzo del prodotto finanziario in modo esatto. Per risolvere questo problema, si può utilizzare l'algoritmo di Monte Carlo per simulare il prezzo del sottostante nel tempo:

1. Si parte da un modello matematico che descrive il comportamento del prezzo del sottostante nel tempo. Ad esempio, si può utilizzare il modello di Black-Scholes, che è uno dei modelli più comuni per descrivere il comportamento del prezzo di un'azione.
2. Si generano numeri casuali che rappresentano le fluttuazioni di prezzo del sottostante nel tempo. Questi numeri casuali sono generati da un generatore di numeri casuali, che produce sequenze di numeri che soddisfano determinate proprietà statistiche; questi numeri casuali sono utilizzati per simulare il comportamento del prezzo del sottostante nel tempo.

3. Si utilizza l'algoritmo di Monte Carlo per simulare il prezzo del prodotto finanziario nel tempo: ad ogni istante di tempo, si valuta se esercitare il prodotto finanziario o meno in base al prezzo del sottostante simulato. Il valore del prodotto finanziario è calcolato come il valore atteso del guadagno del prodotto in ogni istante di tempo.
4. Si ripetono i passaggi 2 e 3 un grande numero di volte, utilizzando diversi numeri casuali in ogni simulazione; in questo modo, si ottiene una stima del valore atteso del prodotto finanziario.
5. Il valore atteso calcolato nell'ultimo passaggio è utilizzato come stima del prezzo del prodotto finanziario.

Anche in questo caso è necessario precisare che l'accuratezza dell'approssimazione dipende dal numero di numeri casuali generati e dalla qualità del generatore di numeri casuali utilizzato; inoltre, l'algoritmo può richiedere un tempo di calcolo elevato, soprattutto se il modello sottostante è molto complesso o se il numero di simulazioni richiesto è elevato.

Ci sono molte librerie Python che implementano l'algoritmo di Monte Carlo per la valutazione di strumenti finanziari:

- **Numpy**: Numpy è una libreria Python che fornisce funzioni per la gestione di array e matrici. È molto utile per la generazione di numeri casuali e per la manipolazione di dati. In particolare, la funzione numpy.random può essere utilizzata per generare numeri casuali necessari all'algoritmo di Monte Carlo.
- **Scipy**: Scipy è una libreria Python che fornisce funzioni per la computazione scientifica. In particolare, la funzione scipy.stats può essere utilizzata per calcolare la distribuzione di probabilità dei dati e per la generazione di numeri casuali.
- **PyMC3**: PyMC3 è una libreria Python che fornisce un'implementazione completa dell'algoritmo di Monte Carlo a Catena di Markov (MCMC). Questo algoritmo è una variante dell'algoritmo di Monte Carlo e viene utilizzato per valutare modelli statistici complessi.
- **QuantLib**: QuantLib è una libreria open-source C++ che fornisce funzioni per la valutazione di strumenti finanziari. Esiste anche una versione Python di QuantLib chiamata PyQL che fornisce funzionalità simili.

3. Esempi

Ecco un esempio di utilizzo di Numpy per generare numeri casuali nell'ambito dell'algoritmo di Monte Carlo in finanza. Supponiamo di voler valutare un'opzione di acquisto su un'azione.

Il prezzo dell'azione segue il modello di Black-Scholes, che può essere espresso come un processo stocastico. Per generare numeri casuali che rappresentano il comportamento del prezzo dell'azione nel tempo, possiamo utilizzare la funzione numpy.random.normal di Numpy.

```
import numpy as np

# Parametri dell'opzione
S0 = 100      # prezzo iniziale dell'azione
K = 110       # prezzo di esercizio dell'opzione
r = 0.05      # tasso di interesse senza rischio
sigma = 0.2   # volatilità dell'azione
T = 1         # tempo di maturità dell'opzione

# Numero di simulazioni
N = 10000

# Generazione dei numeri casuali
dt = 1/252
S = np.zeros((N, 252))
S[:,0] = S0
for i in range(1, 252):
    S[:,i]=S[:,i-1]*np.exp((r-0.5*sigma**2)*dt+
           sigma*np.sqrt(dt)*np.random.normal(size=N))

# Valutazione dell'opzione
discount_factor = np.exp(-r*T)
payoff = np.maximum(S[:, -1] - K, 0)
option_value = discount_factor * np.mean(payoff)

print("Valore dell'opzione:", option_value)
```

In questo esempio, la funzione numpy.random.normal viene utilizzata per generare numeri casuali con media zero e deviazione standard uno. Questi numeri casuali vengono poi utilizzati per simulare il comportamento del prezzo dell'azione nel tempo.

Successivamente, si calcola il guadagno dell'opzione alla data di maturità e si utilizza il valore atteso dei guadagni per valutare l'opzione. Il valore atteso viene calcolato utilizzando la funzione numpy.mean.

Un altro esempio di applicazione dell'algoritmo è quello relativo al **test di primalità**: il test di primalità è utilizzato per determinare se un numero intero positivo è primo o composto.

Esistono molti algoritmi per il test di primalità, ma l'algoritmo di Monte Carlo è uno dei più comuni; tale algoritmo è basato sulla legge di Fermat e utilizza numeri casuali per generare una serie di candidati primi:

1. Si sceglie un numero intero positivo n da testare per la primalità.
2. Si sceglie un numero intero positivo a compreso tra 2 e $n-1$.
3. Si calcola $a^{n-1} \ mod \ n$:
    a. se il risultato è diverso da 1, allora n è composto
    b. in caso contrario, si procede al passaggio successivo.
4. Si sceglie un altro numero a e si ripete il passaggio 3.
5. Si ripete il passaggio 4 un grande numero di volte, utilizzando diversi numeri casuali come a.
6. Se tutti i test hanno dato esito positivo, allora n è probabilmente primo con alta probabilità.

L'algoritmo di Monte Carlo per il test di primalità è veloce e facile da implementare, ma non garantisce l'esattezza della risposta. Tuttavia, aumentando il numero di test e di numeri casuali utilizzati, è possibile aumentare l'accuratezza della risposta.

Python offre diverse librerie per l'implementazione dell'algoritmo di Monte Carlo per il test di primalità, ad esempio, la libreria sympy fornisce la funzione sympy.isprime che utilizza un algoritmo di test di primalità di tipo probabilistico basato sull'algoritmo di Monte Carlo:

```
import random
import sympy

# Numero da testare per la primalità
n = 123456789

# Numero di test da effettuare
k = 10

# Effettua k test di primalità
for i in range(k):
    # Sceglie un numero casuale compreso tra 2 e n-1
    a = random.randint(2, n-1)

    # Calcola a^(n-1) mod n
    r = pow(a, n-1, n)

    # Se il risultato è diverso da 1, allora n è composto
    if r != 1:
        print(n, "è composto")
        break
```

```
# Se tutti i test hanno avuto successo, n è probabilmente primo
else:
    if sympy.isprime(n):
        print(n, "è primo")
    else:
        print(n, "è composto")
```

Occorre notare che si è utilizzata la clausola for else, per cui l'else viene eseguito se il for è eseguito senza che si sia generato un break.

In questo esempio, il numero da testare per la primalità è il numero intero positivo n. Il numero di test di primalità da effettuare è k. Il programma sceglie k numeri casuali compresi tra 2 e $n-1$ e per ogni numero, calcola $a^{n-1} \ mod \ n$. Se il risultato è diverso da 1, allora n è composto. Se tutti i test hanno avuto successo, allora n è probabilmente primo.

La funzione pow viene utilizzata per calcolare $a^{n-1} \ mod \ n$; la funzione sympy.isprime viene utilizzata per verificare se il numero è effettivamente primo.

Si noti che l'algoritmo di Monte Carlo per il test di primalità non garantisce l'esattezza della risposta, ma aumentando il numero di test k e di numeri casuali utilizzati, è possibile aumentare l'accuratezza della risposta.

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.