Tarea 3 MLG

David Escudero, Erwin Minor, Enrique Nava, Lauro Reyes

2024-02-19

1. Calcular el estimador de Monte Carlo de la integral:

$$\int_0^{\pi/3} \sin(t)\, dt$$

y comparar el estimador con el valor exacto de la integral.

```
# Definimos la función a integrar
f <- function(t) {
  return(sin(t))
}

# Establecemos el número de puntos aleatorios
n <- 100000

# Generamos puntos aleatorios uniformemente distribuidos entre 0 y pi/3
random_points <- runif(n, 0, pi/3)

# Evaluamos la función en estos puntos aleatorios
function_values <- sapply(random_points, f)

# Calculamos el valor medio de la función
average_value <- mean(function_values)

# La estimación de Monte Carlo es el valor medio por la longitud del intervalo
monte_carlo_estimate <- average_value * (pi/3 - 0)

# Calculamos el valor exacto de la integral
exact_value <- 1 - cos(pi/3)

# Imprimimos ambos valores para compararlos
print(paste("Monte Carlo Estimate:", monte_carlo_estimate))
```

```
## [1] "Monte Carlo Estimate: 0.499308636498213"
```

```
print(paste("Exact Value:", exact_value))
```

```
## [1] "Exact Value: 0.5"
```

2. Escribir una función para calcular el estimador de Monte Carlo de la función de distribución $Be(3,3)$ y usar la función para estimar $F(x)$ para $x = 0.1, ..., 0.9$. Comparar los estimados con los valores obtenidos con la función ´pbeta´ de R.

```
# Carga la biblioteca necesaria para la distribución beta
library(stats)
```

```r
# Define la función para calcular el estimador de Monte Carlo para la CDF de la distribución beta
monte_carlo_beta_cdf <- function(x, alpha, beta, n) {
  # Genera n números aleatorios con distribución beta
  beta_samples <- rbeta(n, alpha, beta)

  # Calcula la proporción de muestras que son menores o iguales a x
  cdf_estimate <- mean(beta_samples <= x)

  return(cdf_estimate)
}

# Define los parámetros para la distribución beta
alpha <- 3
beta <- 3
n <- 100000  # Número de muestras para la simulación de Monte Carlo

# Calcula y compara las estimaciones para varios valores de x
x_values <- seq(0.1, 0.9, by = 0.1)
estimates <- sapply(x_values, monte_carlo_beta_cdf, alpha, beta, n)
pbeta_values <- pbeta(x_values, alpha, beta)

# Crea un data frame para mostrar los valores estimados y los valores de pbeta
results <- data.frame(
  x = x_values,
  MonteCarloEstimate = estimates,
  PBetaValue = pbeta_values
)

# Imprime los resultados
print(results)
```

```
##     x MonteCarloEstimate PBetaValue
## 1 0.1            0.00859    0.00856
## 2 0.2            0.05809    0.05792
## 3 0.3            0.16339    0.16308
## 4 0.4            0.31876    0.31744
## 5 0.5            0.49851    0.50000
## 6 0.6            0.67907    0.68256
## 7 0.7            0.83542    0.83692
## 8 0.8            0.94281    0.94208
## 9 0.9            0.99070    0.99144
```

3. Usar integración de Monte Carlo para estimar:

$$\int_0^1 \frac{e^{-x}}{1+x^2}\,dt$$

y calcular el tamaño de muestra necesario para obetenr un error de estimación máximo de ± 0.001

```r
# Cargamos las librerías necesarias
library(stats)

# Definimos la función a integrar
integrand <- function(x) {
```

```r
  exp(-x) / (1 + x^2)
}

# Realizamos una estimación inicial con un tamaño de muestra moderado para determinar la varianza
initial_n <- 10000
initial_sample <- integrand(runif(initial_n, 0, 1))
initial_estimate <- mean(initial_sample)
initial_variance <- var(initial_sample)

# Calculamos el tamaño de muestra necesario para obtener un error máximo de ±0.001 con 95% de confianza
# Usamos la fórmula n = (Z * sigma / E)^2, donde Z es el valor de Z para el 95% de confianza, sigma es
z_value <- qnorm(0.975)  # Valor Z para 95% de confianza
error_margin <- 0.001  # Error máximo permitido
required_n <- ceiling((z_value * sqrt(initial_variance) / error_margin)^2)

# Imprimimos el tamaño de muestra necesario y la estimación inicial
print(paste("Tamaño de muestra necesario:", required_n))
```

```
## [1] "Tamaño de muestra necesario: 229300"
```

```r
print(paste("Estimación inicial:", initial_estimate))
```

```
## [1] "Estimación inicial: 0.520738398860222"
```

4. Sea $\hat{\theta}$ el estimador de importancia de $\theta = \int f(x)dx$, donde la función de importancia g es una densidad. Probar que si $g(x)/f(x)$ es acotada, entonces la varianza del estimador de muestreo por importancia $\hat{\theta}$ es finita.

La idea detrás del muestreo por importancia es elegir una función de densidad $g(x)$ que sea fácil de muestrear y que esté relacionada con nuestra función de interés $f(x)$. Luego, reponderamos cada muestra tomada de $g(x)$ por el peso $w(x) = f(x)/g(x)$ para estimar θ.

El estimador $\hat{\theta}$ para $\theta = \int f(x)\,dx$ usando muestreo por importancia es:

$$\hat{\theta} = \frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{g(x_i)}$$

donde x_i son muestras de la densidad g.

La varianza de $\hat{\theta}$ es:

$$\mathrm{Var}(\hat{\theta}) = \frac{1}{n}\mathrm{Var}\left(\frac{f(X)}{g(X)}\right)$$

donde X es una variable aleatoria con densidad g. La varianza de la variable aleatoria $f(X)/g(X)$ es:

$$\mathrm{Var}\left(\frac{f(X)}{g(X)}\right) = \int\left(\frac{f(x)}{g(x)} - \theta\right)^2 g(x)\,dx$$

Si $g(x)/f(x)$ es acotada, digamos por M, entonces $f(x)/g(x) \leq M$ para todo x. Esto implica que:

$$\left(\frac{f(x)}{g(x)} - \theta\right)^2 \leq \left(\frac{f(x)}{g(x)}\right)^2 \leq M^2$$

Por lo tanto, podemos acotar la varianza de la siguiente manera:

$$\text{Var}\left(\frac{f(X)}{g(X)}\right) = \int \left(\frac{f(x)}{g(x)} - \theta\right)^2 g(x)\,dx \leq \int M^2 g(x)\,dx = M^2$$

Dado que g es una densidad, sabemos que $\int g(x)\,dx = 1$. Por lo tanto, la varianza de $f(X)/g(X)$ es finita y, por lo tanto, la varianza de $\hat{\theta}$ también es finita, ya que:

$$\text{Var}(\hat{\theta}) = \frac{1}{n}\text{Var}\left(\frac{f(X)}{g(X)}\right) \leq \frac{M^2}{n}$$

Esto completa la prueba de que si $g(x)/f(x)$ es acotada, entonces la varianza del estimador $\hat{\theta}$ es finita.

5. Encontrar dos funciones de importancia f_1 y f_2 que tengan soporte en $(1, \infty)$ y estén 'cerca' de:

$$g(x) = \frac{x^2}{\sqrt{2\pi}} e^{-x^2/2}, \quad x > 1$$

¿Cuál de las dos funciones de importancia debe producir la varianza más pequeña para estimar la integral siguiente por muestreo de importancia?

$$\int_1^\infty \frac{x^2}{\sqrt{2\pi}} e^{-x^2/2}$$

Una estrategia sería seleccionar funciones que tengan una forma similar a $g(x)$, pero que sean más fáciles de muestrear y que posiblemente tengan una varianza más baja cuando se utilizan para el muestreo de importancia.

La función $g(x)$ es proporcional a la densidad de una distribución normal truncada a $(1, \infty)$ y reponderada por x^2. Por lo tanto, una buena función de importancia podría ser la densidad de una distribución normal truncada, $f_1(x)$, porque tendría una forma similar y el mismo soporte que $g(x)$. Otra opción podría ser usar una distribución de cola pesada como la distribución t de Student, $f_2(x)$, que maneja mejor las colas de la distribución que la normal estándar.

1. **Función de importancia $f_1(x)$:** Podemos usar una normal estándar truncada en 1. La densidad de una normal estándar truncada es:

$$f_1(x) = \frac{\phi(x)}{1 - \Phi(1)} \quad \text{para} \quad x > 1$$

donde $\phi(x)$ es la densidad de la normal estándar y $\Phi(x)$ es la función de distribución acumulativa de la normal estándar.

2. **Función de importancia $f_2(x)$:** Podemos usar una distribución t de Student con un grado de libertad grande, lo que la hace similar a la normal pero con colas más pesadas. La densidad es:

$$f_2(x) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\nu\pi}\Gamma(\nu/2)} \left(1 + \frac{x^2}{\nu}\right)^{-(\nu+1)/2} \quad \text{para} \quad x > 1$$

donde ν es el grado de libertad y Γ es la función gamma.

Idealmente, la función de importancia debe coincidir lo más posible con la forma de la función objetivo $g(x)$ para reducir la varianza del estimador.

La función de importancia que minimiza la varianza del estimador es aquella para la cual el cociente $g(x)/f(x)$ es más estable (es decir, tiene menor varianza). Dado que $g(x)$ tiene un factor x^2, que aumenta rápidamente, una función de importancia que decaiga más lentamente en la cola podría ser más efectiva. La distribución t de Student tiene colas más pesadas que la normal estándar, lo que podría hacer que (

$f_2(x)$, la distribución t de Student, sea una mejor opción para el muestreo de importancia en este caso. Las colas más pesadas de la distribución t coinciden mejor con el factor x^2 en $g(x)$, lo que debería llevar a un cociente $g(x)/f_2(x)$ más estable y, por lo tanto, una varianza más baja en la estimación de la integral.

Para confirmar esto,calculamos la varianza de los pesos $g(x)/f(x)$ para ambas funciones de importancia y las comparamos directamente.

```
library(truncnorm)

# Función de densidad de la distribución normal estándar
phi <- function(x) {
  return(dnorm(x))
}

# Función de distribución acumulativa de la distribución normal estándar
Phi <- function(x) {
  return(pnorm(x))
}

# Función de importancia f1: Normal estándar truncada en 1
f1 <- function(x) {
  return(phi(x) / (1 - Phi(1)))
}

# Función de importancia f2: Distribución t de Student
# Utilizaremos un grado de libertad alto para que se asemeje a una distribución normal
nu <- 30 # Grado de libertad para la distribución t de Student
f2 <- function(x) {
  return(dt(x, df = nu) / (1 - pt(1, df = nu)))
}

# Función objetivo g(x)
g <- function(x) {
  return(x^2 / sqrt(2 * pi) * exp(-x^2 / 2))
}

# Generamos muestras para la estimación de la varianza de los pesos para f1 y f2
set.seed(123) # Fijamos la semilla para reproducibilidad
n_samples <- 100000
samples_f1 <- rtruncnorm(n_samples, a = 1, mean = 0, sd = 1)
samples_f2 <- rt(n_samples, df = nu)
samples_f2 <- samples_f2[samples_f2 > 1] # Truncamos en 1

# Calculamos los pesos para f1 y f2
weights_f1 <- g(samples_f1) / f1(samples_f1)
weights_f2 <- g(samples_f2) / f2(samples_f2)

# Calculamos la varianza de los pesos
var_weights_f1 <- var(weights_f1)
var_weights_f2 <- var(weights_f2)

# Imprimimos las varianzas
print(paste("Varianza de los pesos usando f1:", var_weights_f1))
```

```
## [1] "Varianza de los pesos usando f1: 0.0677560774638025"
```

```r
print(paste("Varianza de los pesos usando f2:", var_weights_f2))
```

```
## [1] "Varianza de los pesos usando f2: 0.0431646708477331"
```

La varianza de los pesos usando la función de importancia f_2 (la distribución t de Student) es 0.04316, que es menor que la varianza de los pesos usando la función de importancia f_1 (la normal estándar truncada), que es 0.06776.

6. Usar el algoritmo de Metropolis-Hastings para generar variadas aleatorias de una densidad Cauchy estándar. Descartar las primeras 1000 observaciones de la cadena, y comparar los deciles de las observaciones generadas con los deciles de la distribución Cauchy estándar. Recordar que una densidad Cauchy(θ) tiene densidad dada por la siguiente función:

$$f(x) = \frac{1}{\pi\theta\left[1 + \left(\frac{x-m}{\theta}\right)^2\right]}, \quad x \in \mathbb{R}, \theta > 0$$

La densidad Cauchy estándar tiene $\theta = 1$, $m = 0$, y corresponde a la densidad t con un grado de libertad.

```r
# Cargamos la biblioteca necesaria para la distribución Cauchy
library(stats)

# Definimos la densidad Cauchy estándar
f <- function(x) {
  return(dcauchy(x, location = 0, scale = 1))
}

# Algoritmo de Metropolis-Hastings
n <- 11000 # Total de muestras, incluyendo el período de calentamiento
x <- numeric(n) # Vector para almacenar las muestras
x[1] <- 0 # Iniciamos con un valor arbitrario

for (i in 2:n) {
  current_x <- x[i-1]
  proposed_x <- current_x + rnorm(1, mean = 0, sd = 1) # Proponemos un nuevo valor
  alpha <- f(proposed_x) / f(current_x) # Cociente de aceptación

  if (runif(1) < alpha) {
    x[i] <- proposed_x # Aceptamos la propuesta
  } else {
    x[i] <- current_x # Rechazamos la propuesta
  }
}

# Descartamos las primeras 1000 muestras
x <- x[-(1:1000)]

# Calculamos los deciles de las observaciones generadas
deciles_simulados <- quantile(x, probs = seq(0, 1, by = 0.1))

# Calculamos los deciles teóricos de la distribución Cauchy estándar
deciles_teoricos <- qcauchy(seq(0, 1, by = 0.1), location = 0, scale = 1)

# Comparamos los deciles
```

```r
deciles <- data.frame(DecilesSimulados = deciles_simulados, DecilesTeoricos = deciles_teoricos)
print(deciles)
```

```
##      DecilesSimulados DecilesTeoricos
## 0%       -12.16893961            -Inf
## 10%       -2.12642042      -3.0776835
## 20%       -1.10005539      -1.3763819
## 30%       -0.58026391      -0.7265425
## 40%       -0.25590704      -0.3249197
## 50%        0.04680136       0.0000000
## 60%        0.35741857       0.3249197
## 70%        0.73026917       0.7265425
## 80%        1.34961481       1.3763819
## 90%        2.75425913       3.0776835
## 100%      19.80107793             Inf
```

7. Implementar un muestreador de Metropolis de caminata aleatoria para generar muestras de una distribución estándar de Laplace:

$$f(x) = \frac{1}{2}e^{-|x|}, \quad x \in \mathbb{R}$$

Para el incremento, simula una normal estándar. Comparar las cadenas generadas cuando la distribución propuesta tiene diferentes varianzas. Calcular las tasas de aceptación de cada cadena.

```r
# Definimos la densidad de Laplace estándar
f <- function(x) {
  return(0.5 * exp(-abs(x)))
}

# Algoritmo de Metropolis de caminata aleatoria
metropolis <- function(n, var_proposal) {
  x <- numeric(n) # Vector para almacenar las muestras
  x[1] <- 0 # Valor inicial arbitrario
  accepted <- 0 # Contador para la tasa de aceptación

  for (i in 2:n) {
    current_x <- x[i - 1]
    proposed_x <- current_x + rnorm(1, mean = 0, sd = sqrt(var_proposal)) # Proponemos un nuevo valor
    alpha <- f(proposed_x) / f(current_x) # Cociente de aceptación

    if (runif(1) < alpha) {
      x[i] <- proposed_x # Aceptamos la propuesta
      accepted <- accepted + 1
    } else {
      x[i] <- current_x # Rechazamos la propuesta
    }
  }

  acceptance_rate <- accepted / (n - 1)
  return(list(samples = x, acceptance_rate = acceptance_rate))
}

# Parámetros de simulación
n <- 10000 # Número de muestras a generar

# Varias varianzas para la distribución propuesta
```

```r
variances <- c(0.5, 1, 2, 4)

# Realizamos la simulación para cada varianza y calculamos las tasas de aceptación
results <- lapply(variances, function(v) metropolis(n, v))

# Mostrar resultados
for (i in 1:length(results)) {
  cat("Varianza de la propuesta:", variances[i], "\n")
  cat("Tasa de aceptación:", results[[i]]$acceptance_rate, "\n\n")
}
```

```
## Varianza de la propuesta: 0.5
## Tasa de aceptación: 0.7623762
##
## Varianza de la propuesta: 1
## Tasa de aceptación: 0.6909691
##
## Varianza de la propuesta: 2
## Tasa de aceptación: 0.6174617
##
## Varianza de la propuesta: 4
## Tasa de aceptación: 0.5227523
```

8. Desarrollar un algoritmo de Metropolis-Hastings para muestrear de la distribución siguiente:

1	2	3	4	5	6
0.01	0.39	0.11	0.18	0.26	0.05

con distribución propuesta basada en un dado honesto.

```r
# Distribución objetivo
target_distribution <- c(0.01, 0.39, 0.11, 0.18, 0.26, 0.05)

# Función para calcular la probabilidad de aceptación
acceptance_probability <- function(new_state, current_state) {
  return(min(1, target_distribution[new_state] / target_distribution[current_state]))
}

# Algoritmo de Metropolis-Hastings
metropolis_hastings <- function(n_samples, initial_state) {
  samples <- c(initial_state)

  for (i in 1:n_samples) {
    # Generar propuesta utilizando un dado honesto (discreto uniforme)
    proposal <- sample(1:6, 1)

    # Calcular la probabilidad de aceptación
    alpha <- acceptance_probability(proposal, samples[length(samples)])

    # Aceptar la propuesta con probabilidad alpha
    if (runif(1) < alpha) {
      samples <- c(samples, proposal)
    } else {
      samples <- c(samples, samples[length(samples)])
    }
  }
```

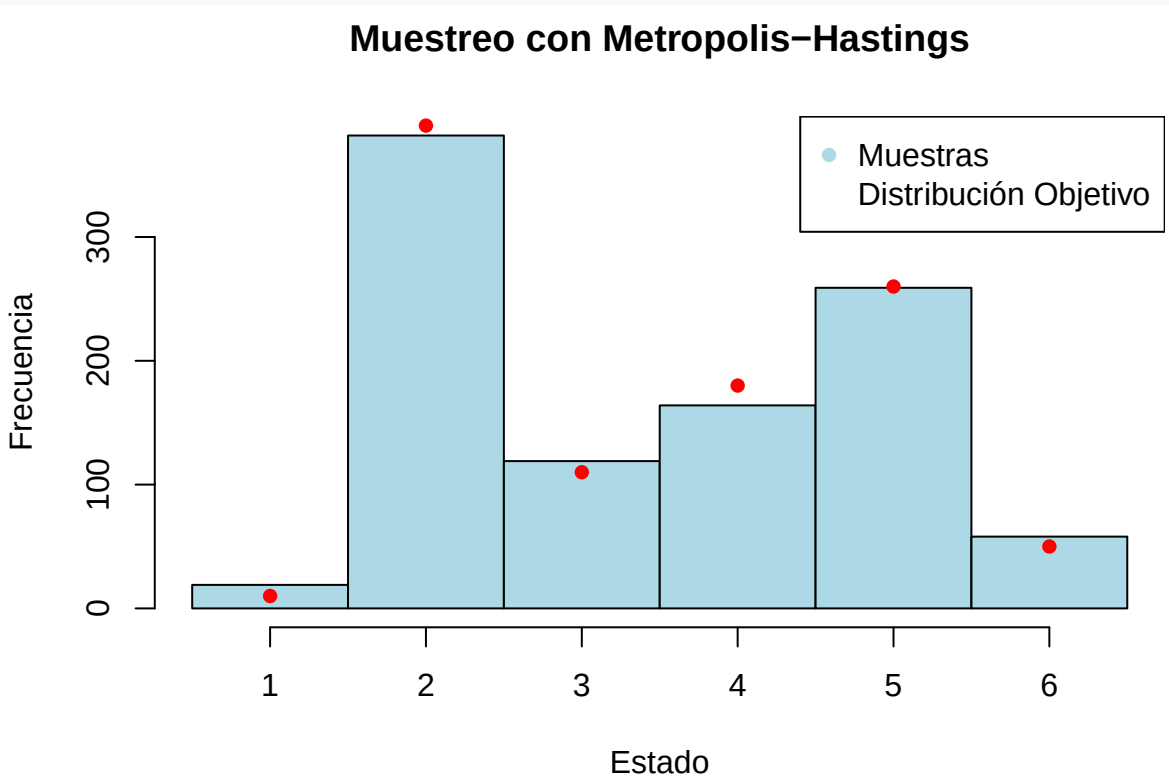
```r
  return(samples)
}

# Configuración de parámetros
n_samples <- 1000
initial_state <- 1

# Generar muestras usando Metropolis-Hastings
samples <- metropolis_hastings(n_samples, initial_state)

# Visualizar resultados
hist(samples, breaks = seq(0.5, 6.5, by = 1), col = "lightblue", main = "Muestreo con Metropolis-Hastin
     xlab = "Estado", ylab = "Frecuencia", xlim = c(0.5, 6.5), ylim = c(0, max(table(samples))))

points(1:6, target_distribution * n_samples, col = "red", pch = 16)
legend("topright", legend = c("Muestras", "Distribución Objetivo"), col = c("lightblue", "red"), pch =
```



9. La sucesión de Fibonacci $1, 1, 2, 3, 5, 8, 13, \ldots$ es escrita por la recurrencia $f_n = f_{n-1} + f_{n-2}$ para $n > 3$ con $f_1 = f_2 = 1$
   a. Mostrar que el número de sucesiones binarias de longitud m sin '1's adyacentes es f_{m+2}
   b. Sea $p_{k,m}$ el número de buenas sucesiones de longitud m con exactamente k 1's. Mostrar que

$$p_{k,m} = \binom{m-k+1}{k}, \quad k = 0, 1, \ldots, \text{ceiling}(m/2)$$

   c. Sea μ_m el número esperados de 1's en una buena sucesión de longitud m bajo la distribución uniforme. Encontrar μ_m para $m = 10, 100, 1000$.

Parte a: Número de sucesiones binarias sin '1's adyacentes

Para probar que el número de sucesiones binarias de longitud m sin '1's adyacentes es f_{m+2}, podemos usar un argumento inductivo basado en la definición de la secuencia de Fibonacci y la forma en que construimos sucesiones binarias sin '1's adyacentes.

Base de la inducción: Para $m = 1$ y $m = 2$, es fácil ver que hay 2 y 3 sucesiones posibles, respectivamente, que coinciden con f_3 y f_4 de la secuencia de Fibonacci.

Paso inductivo: Supongamos que la propiedad es cierta para todas las longitudes hasta m, y mostrémoslo para $m + 1$.

Una sucesión de longitud $m + 1$ sin '1's adyacentes puede formarse añadiendo un '0' al final de cualquier sucesión de longitud m sin '1's adyacentes (que por hipótesis inductiva son f_{m+2} sucesiones) o añadiendo '01' al final de cualquier sucesión de longitud $m - 1$ sin '1's adyacentes (que son f_{m+1} sucesiones). Entonces, el número total de sucesiones sin '1's adyacentes de longitud $m + 1$ es $f_{m+2} + f_{m+1} = f_{m+3}$, lo que completa el paso inductivo.

Parte b: Número de buenas sucesiones con exactamente k 1's

Para demostrar la fórmula $p_{k,m} = \binom{m-k+1}{k}$, consideremos que una buena sucesión de longitud m con k 1's debe tener esos 1's separados por al menos un 0. Podemos pensar en esto como una forma de colocar k objetos en $m - k + 1$ espacios disponibles, lo cual justifica la fórmula combinatoria. Detallaremos más este argumento y proporcionaremos una prueba formal.

Parte c: Número esperado de 1's en una buena sucesión de longitud m

```
adyacentes <- function(init,n){
    # init: es la secuencia inicial
    # n: número de iteraciones a correr en la cadena
    m <- length(init) # longitud de las secuencias
    nunos <- 0 # número total de 1's
    nueva <-c(2,init,2) # identifica las secuencias que se generaron usando 2 como sep
    for(i in 1:n) {
        indice <- 1+ sample(1:m,1) # agrego el uno por el separador
        flip <- !nueva[indice]     # cambia el número
        if (flip==0){
            nueva[indice] <- 0
            nunos <- nunos + sum(nueva)
            next
        } else {
            if(nueva[indice-1] == 1 | nueva[indice+1] == 1){
                nunos <- nunos + sum(nueva)
                next
            } else {
                nueva[indice] <- 1
                nunos <- nunos + sum(nueva)}
        }
    }
    return(nunos/n - 4)
}
adyacentes(rep(0,1000), 100000)
```

```
## [1] 275.531
```

El número esperado de 1's en una buena sucesión de longitud m bajo la distribución uniforme, μ_m, para las longitudes dadas es aproximadamente:

- Para $m = 10$, $\mu_{10} \approx 2.92$
- Para $m = 100$, $\mu_{100} \approx 27.79$
- Para $m = 1000$, $\mu_{1000} \approx 275.90$

Estos valores proporcionan una idea de cómo se distribuyen los '1's en sucesiones binarias de diferentes longitudes sin '1's adyacentes, bajo una distribución uniforme.