Συναρτήσεις

Συμπληρωματικές σημειώσεις

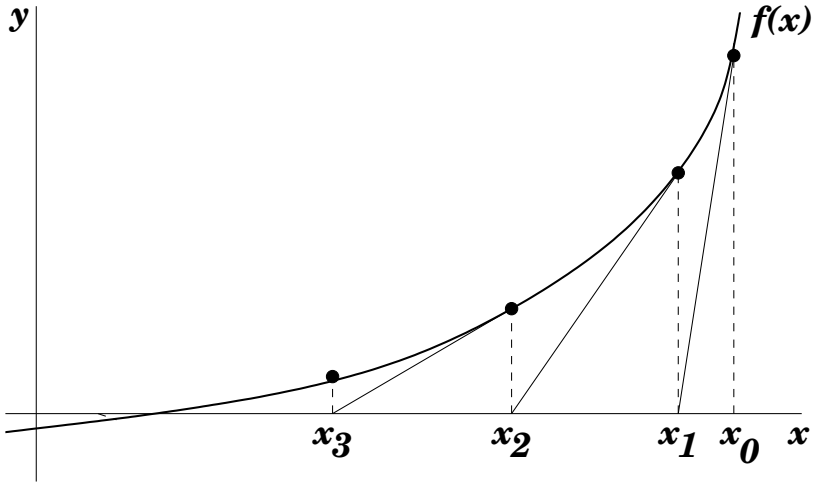
Μιχάλης Δρακόπουλος

Υπολογισμός τετραγωνικής ρίζας

Μέθοδος Newton

Ο υπολογισμός της τετραγωνικής ρίζας $x = \sqrt{\alpha}$ ισοδυναμεί με την εύρεση της θετικής ρίζας της συνάρτηση $f(x) = x^2 - \alpha$.

Θα χρησιμοποιήσουμε την μέθοδο Newton που είναι μια *αριθμητική* μέθοδος εύρεσης ριζών πραγματικών συναρτήσεων. Η μέθοδος Newton υπολογίζει *επαναληπτικά* διαδοχικές προσεγγίσεις στη ρίζα μιας συνάρτησης $f(x)$. Ξεκινάμε με μια αρχική εκτίμηση x_0 της ρίζας. Υπό κατάλληλες προϋποθέσεις και εφόσον η x_0 είναι σχετικά κοντά στη ρίζα, υπολογίζουμε την x_1 η οποία είναι μια *καλύτερη* προσέγγιση της ρίζας από τη x_0.



Γεωμετρικά η x_1 είναι το σημείο τομής της εφαπτομένης της f στο x_0 με τον άξονα των x.

$$\frac{f(x_0)}{x_0 - x_1} = f'(x_0) \Rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Η διαδικασία επαναλαμβάνεται έως ότου πλησιάσουμε τη ρίζα της f στα πλαίσια της επιθυμητής ακρίβειας. Το γενικό βήμα της μεθόδου είναι:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Για $f(x) = x^2 - \alpha$:

$$x_{k+1} = x_k - \frac{x_k^2 - \alpha}{2x_k} = x_k - \frac{x_k - \alpha/x_k}{2} = \frac{x_k + a/x_k}{2}$$

Προφανώς δεν χρειάζονται δυο ξεχωριστές μεταβλητές για τα x_k και x_{k+1}.

Θεωρούμε ότι *προσεγγίσαμε* ικανοποιητικά τη ρίζα όταν φράξουμε το *σχετικό σφάλμα* από κάποιον πολύ μικρό αριθμό ε (επιθυμητή ακρίβεια). Δηλαδή όταν

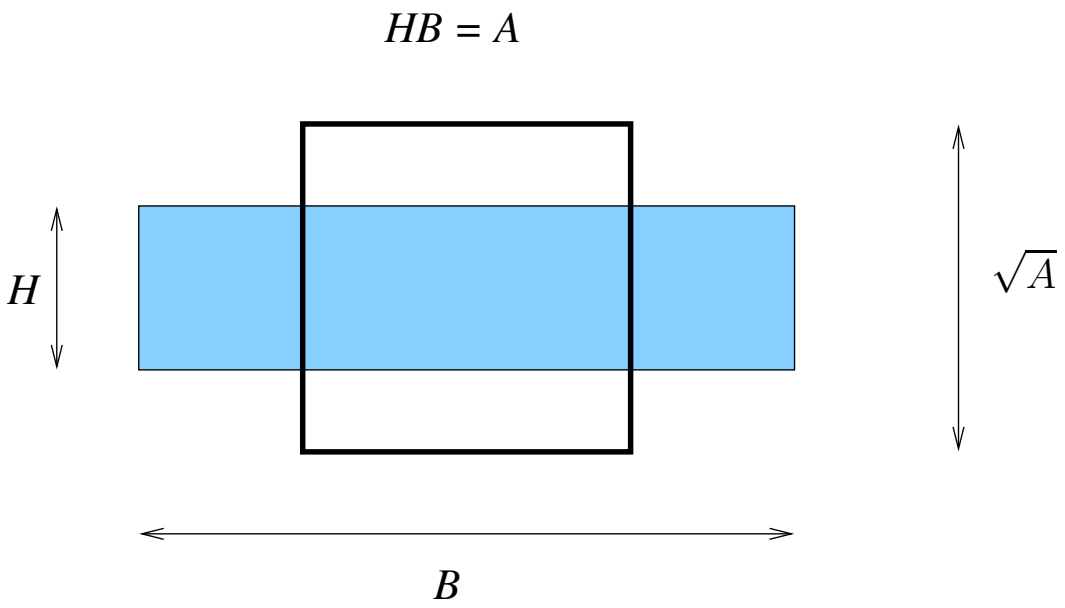
$$\frac{|x^2 - \alpha|}{x^2} = \frac{|x - \alpha/x|}{|x|} < \varepsilon.$$

```
def newton_sqrt(a, epsilon=1e-8):
    # Τετραγωνική ρίζα με τη μ. Newton
    if a == 0:
        x = 0
    else:
        x = a                    # αρχική εκτίμηση
        while abs(x - a/x) > epsilon*x:
            x = (x + a/x)/2
    return x
```

Μια γεωμετρική προσέγγιση

Επαναλαμβάνουμε σε συντομία τον αλγόριθμο που είδαμε στο πρώτο μέρος των σημειώσεων.

Η $\sqrt{A}$ είναι το μήκος πλευράς τετραγώνου με εμβαδόν A.



Το γεγονός αυτό μας οδηγεί στον εξής αλγόριθμο υπολογισμού της $\sqrt{A}$:

- Έστω ορθογώνιο εμβαδού A, με $H = 1$ και $B = A$.
- Κατασκεύασε διαδοχικά ορθογώνια εμβαδού A που να *προσεγγίζουν* όλο και περισσότερο το τετράγωνο, θέτοντας:
$$B = (B + H)/2, \quad H = A/B$$
- Σταμάτησε όταν κάποιο ορθογώνιο *μοιάζει αρκετά* με τετράγωνο.

```
def geom_sqrt(A, epsilon=1e-8):
    if A == 0:
        B = 0
    else:
        H = 1; B = A
        while abs(B - H) > epsilon*B:
            B = (B + H)/2
            H = A / B
    return B
```

Ο αλγόριθμος αυτός *ταυτίζεται* με τη μέθοδο Newton για $x_0 = a$.

Μέθοδος Taylor

Ανάπτυγμα σειράς

Ένα από τα παράδοξα του Ζήνωνα έχει να κάνει με τη διχοτόμηση μιας απόστασης.

Αυτό μεταφράζεται στον υπολογισμό του αθροίσματος:

$$s = 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + \dots$$

Έχουμε δηλαδή το άθροισμα *απείρων θετικών* όρων που όμως είναι *πεπερασμένο*:

$$2s = 1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + \dots$$

δηλαδή

$$2s = 1 + s$$

επομένως το άθροισμα της σειρά συγκλίνει στο $s = 1$.

Ο γενικός όρος της σειράς είναι:

$$t_i = 1/2^i$$

είναι δηλαδή μια *δυναμοσειρά*.

Οι δυναμοσειρές είναι σημαντικές για τα υπολογιστικά μαθηματικά. Σε αλγόριθμους υπολογισμού δυναμοσειρών

αναζητούμε μια σχέση που δίνει τον i-οστό όρο από τον όρο $i-1$.

Για την παραπάνω δυναμοσειρά $s = \sum 1/2^i$, κάθε νέος όρος προκύπτει από τον προηγούμενο:

$$t_i = \frac{1}{2^i} = \frac{1}{2^{i-1}}\frac{1}{2} = t_{i-1}\frac{1}{2}$$

Ενας *ψευδο*-αλγόριθμος που θα υπολόγιζε το άθροισμα *απείρων* όρων της σειράς θα ήταν

```
Sum = 0
term = 0.5
while True:
    Sum += term
    term /= 2
```

Προφανώς κάτι τέτοιο δεν θα μπορούσε να σταθεί ως πρόγραμμα. Παρατηρούμε όμως ότι η σειρά είναι φθίνουσα και από κάποιο σημείο και μετά οι όροι γίνονται τόσο μικροί που το `Sum` δεν μεταβάλεται, λόγω της ακρίβειας της υπολογιστικής αναπαράστασης των πραγματικών αριθμών.

Έτσι ένα κριτήριο τερματισμού των επαναλήψεων είναι ο έλεγχος:

```
Sum == Sum + term
```

Η πρόσθεση του `term` δεν αλλάζει την τιμή του `Sum` και το πρόγραμμα για τη σειρά του Ζήνωνα είναι:

```
Sum = 0
term = 0.5
while Sum != Sum + term:
    Sum += term
    term /= 2
print(f'Zeno sum = {Sum}')
```

Εναλλακτικά θα μπορούσαμε να σταματήσουμε τις επαναλήψεις όταν `term < epsilon`, όταν δηλαδή οι όροι αρχίσουν να γίνονται μικρότεροι από κάποια επιθυμητή ακρίβεια `epsilon`.

Υπολογισμός τετραγωνικής ρίζας - Σειρά Taylor

Διαφορίσιμες συναρτήσεις προσεγγίζονται στην περιοχή κάποιου θ από το ανάπτυγμα Taylor:

$$f(x) = f(\theta) + f'(\theta)(x-\theta) + f''(\theta)\frac{(x-\theta)^2}{2!} + f'''(\theta)\frac{(x-\theta)^3}{3!} + \cdots + f^{(n)}(\theta)\frac{(x-\theta)^n}{n!}$$

Για την τετραγωνική ρίζα: $f(x) = x^{1/2}$ έχουμε:

$f'(x)$	$f''(x)$	$f'''(x)$	$\ldots$
$\frac{1}{2}x^{-1/2}$	$-\frac{1}{4}x^{-3/2}$	$\frac{3}{8}x^{-5/2}$	$\ldots$

Ο τύπος του Taylor υπολογίζει παραγώγους της f στο θ. Για ευκολία, έστω $\theta = 1$, τότε:

$f(1)$	$f'(1)$	$f''(1)$	$f'''(1)$	$\ldots$
1	$\frac{1}{2}$	$-\frac{1}{4}$	$\frac{3}{8}$	$\ldots$

Με βάση τα παραπάνω, η τετραγωνική ρίζα κοντά στο θ προσεγγίζεται από Taylor:

$$\sqrt{x} = 1 + \frac{1}{2}(x-1) - \frac{1}{4}\frac{(x-1)^2}{2!} + \frac{3}{8}\frac{(x-1)^3}{3!} + \cdots = \sum_0^\infty t_i$$

Οι διαδοχικοί όροι της σειράς είναι:

t_0	t_1	t_2	t_3	$\ldots$
1	$\frac{1}{2}(x-1)$	$-\frac{1}{4}\frac{(x-1)^2}{2!}$	$\frac{3}{8}\frac{(x-1)^3}{3!}$	$\ldots$

Είναι γενικά της μορφής

```
coeff * (xpower / factorial)
```

Η σχέση που συνδέει διαδοχικούς όρους, που δίνουν δηλαδή τα νέα `coeff`, `xpower`, `factorial` για τον όρο $i+1$ από τις αντίστοιχες προηγούμενες τιμές για τον όρο i είναι:

```
coeff *= (0.5 - i)
xpower *= (x - 1)
factorial *= (i+1)
```

και η συνάρτηση υπολογισμού της τετραγωνικής ρίζας με τον αλγόριθμο Taylor είναι:

```
def t_sqrt(x):
    term = factorial = coeff = xpower = 1
    t_sum = 0
    i = 0
    while t_sum != t_sum + term:
        t_sum += term
        # Προετοιμασία όρου i+1
        coeff *= (0.5 - i)
        xpower *= (x - 1)
        factorial *= (i + 1)
        term = coeff * xpower / factorial
        i += 1
    return t_sum
```

Μαθηματικά υπάρχει πρόβλημα ως προς τη γενικότητα του παραπάνω αλγορίθμου, καθώς για $\theta = 1$ η $\sqrt{x}$ από τον τύπο του Taylor *συγκλίνει MONO* για:

$$0 < x < 2$$

Για να γενικεύσουμε τον αλγόριθμο, αναγάγουμε *οποιοδήποτε* x στην περιοχή σύγκλισης *παρατηρώντας* ότι

$$\sqrt{4x} = 2\sqrt{x}.$$

Δηλαδή μετά διαδοχικές διαιρέσεις του x με το 4, υπολογίζεται η $\sqrt{x}$ στο διάστημα $(0, 2)$ από τον παραπάνω κώδικα. Ακολουθεί διόρθωση του αποτελέσματος με αντίστοιχο αριθμό πολλαπλασιασμών $\times 2$.

```
def taylor_sqrt(x):
    if x == 0:
        return 0
    correction = 1
    while x >= 2:
        x /= 4
        correction *= 2
    return t_sqrt(x) * correction
```

Έλεγχος πρώτων αριθμών

Ένα βασικό πρόβλημα στη Θεωρία Αριθμών είναι ο έλεγχος για το αν κάποιος ακέραιος είναι πρώτος ή όχι. Οι πρώτοι αριθμοί είναι σημαντικοί σε περιοχές όπως η κρυπτογραφία, η (απο-)κωδικοποίηση,

οι ηλεκτρονικές επικοινωνίες κ.α. και είναι σημαντικό οι αλγόριθμοι ελέγχου να είναι γρήγοροι και αποτελεσματικοί.

Ένας εξαντλητικός αλγόριθμος

Είναι ο n πρώτος?

- Καταμέτρηση όλων των διαιρετών του n (χρειάζεται να υπολογισθούν). Αν είναι ακριβώς 2 τότε ο n είναι πρώτος.
- Οι πιθανοί διαιρέτες του n είναι ≤ n, άρα αρκεί να ελεγχθεί ποιοί από τους $1, 2, \ldots, n$ είναι διαιρέτες του n.

```
def is_prime1(n):
    num_div = 0
    for k in range(1, n+1):
        if n%k == 0:
            num_div += 1
    return num_div == 2
```

Μια **προγραμματιστική παρατήρηση**: το πρόγραμμα επιστρέφει την τιμή της λογικής έκφρασης `num_div == 2`, δηλαδή επιστρέφει `True` όταν βρίσκει *ακριβώς* 2 διαιρέτες, και `False` στην αντίθετη περίπτωση. Αυτό θα μπορούσε να γίνει προφανώς και με μια εντολή `if-else`.

Ο αλγόριθμος αυτός προγραμματίζεται εύκολα, αλλά δεν είναι πρακτικός για μεγάλους αριθμούς: ο 1000005 φαίνεται ότι *δεν* είναι πρώτος στον έλεγχο με το 5. Άρα ο αλγόριθμος μμπορεί να τερματίζει μόλις βρεθεί διαιρέτης > 1.

Ένας γρηγορότερος αλγόριθμος σε 2 παραλλαγές

Μπορούμε να κατασκευάσουμε γρηγορότερους αλγόριθμους παρατηρώντας ότι:

- Περιττεύει η εξέταση πιθανών διαιρετών που είναι *άρτιοι* και > 2: αν ο n δεν διαιρείται από το 2, τότε δεν διαιρείται από κανένα άρτιο.
- Αριθμοί $> \sqrt{n}$ δεν χρειάζεται να ελεγχθούν ως πιθανοί διαιρέτες: αν $\delta_1 \neq delta_2$ είναι δυο διαιρέτες του n τέτοιοι ώστε $\delta_1 \times \delta_2 = n$, τότε *ένας* από τους δ_1, δ_2 θα είναι *οπωσδήποτε* $< \sqrt{n}$ (απόδειξη με απαγωγή σε άτοπο).

Λαμβάνοντας υπόψη τα παραπάνω έχουμε:

```
import math

def is_prime2(n):
    if (n%2 == 0 and n > 2) or n <= 1:
        return False
    else:
        for divisor in range(3, int(math.sqrt(n))+1, 2):
            if n%divisor == 0:
                return False
            else:
```

```
            divisor += 2
        return True
```

Ο λόγος που ελέγχουμε μέχρι και το $\sqrt{n}+1$ είναι για να αποφύγουμε τυχόν σφάλματα που μπορεί να οφείλονται στην αριθμητική των υπολογιστών. Π.χ. για το 49 με διαιρέτες τους 1, 7 και 49, η `sqrt(49)` μπορεί να υπολογίζεται σαν `6.9999999` σε κάποιον υπολογιστή. Το 7 δεν θα ελεγχθεί σαν πιθανός διαιρέτης και ο παραπάνω αλγόριθμος θα έδινε ότι ο 49 είναι πρώτος. Άρα για ασφάλεια, χρειάζεται έλεγχος μέχρι `sqrt(n) + 1`.

Η παρακάτω παραλλαγή αποφεύγει τον υπολογισμό της τετραγωνικής ρίζας.

```
def is_prime3(n):
    if (n%2 == 0 and n > 2) or n <= 1:
        return False
    else:
        divisor = 3
        while divisor*divisor <= n:
            if n%divisor == 0:
                return False
            else:
                divisor += 2
        return True
```

Ένας αποτελεσματικότερος αλγόριθμος

Στην αριθμητική modulo 6 όλοι οι ακέραιοι αριθμοί είναι της μορφής $6k+r,\ $ με $r \in \{0,1,2,3,4,5\}$. Από αυτούς οι πιθανοί διαιρέτες του n θα έχουν $r \in \{1,5\}$. Οι αριθμοί $6k, 6k+2, 6k+3, 6k+4$ διαιρούνται με το 2 ή με το 3.

Επομένως ελέγχουμε αρχικά αν ο n διαιρείται με το 2 ή το 3 και στη συνέχεια ελέγχουμε μέχρι το $\sqrt{n}$ για πιθανούς διαιρέτες του, της μορφής $6k+1$ και $6k+5$. Αν εξαιρέσουμε τη μονάδα ($6\times 0+1$), δοκιμάζουμε διαίρεση με τους αριθμούς $(5,7),(11,13),(17,19),\ldots$. Έχουμε χωρίσει τους αριθμούς σε ομάδες των 2. Οι αριθμοί σε κάθε ομάδα διαφέρουν κατά 2, και οι ομάδες απέχουν μεταξύ τους κατά 6.

Με βάση τα παραπάνω έχουμε:

```
def is_prime4(n):
    if n < 2:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    divisor = 5
    while divisor * divisor <= n:
        if n % divisor == 0 or n % (divisor + 2) == 0:
            return False
        divisor += 6
    return True
```

Υπολογισμός τιμής και παραγώγου μαθηματικής συνάρτησης

Οι συναρτήσεις στην Python είναι **αντικείμενα** και έχουν τα χαρακτηριστικά και τις ιδιότητες που διαθέτουν και οι άλλοι τύποι αντικειμένων. Συγκεκριμένα:

- Μπορούν να εκχωρηθούν σε μεταβλητές

```
def prod(a, b):
    return a*b

x = prod(2, 5)
print(x)  # τυπώνει 10

times = prod  # η συνάρτηση prod "εκχωρείται" στη μτβ times
y = times(2, 5)
print(y)  # τυπώνει πάλι 10
```

  Δηλαδή με το όνομα `times` ουσιαστικά καλούμε την `prod`.

- Μπορούν να είναι ορίσματα άλλων συναρτήσεων. **Παράδειγμα:** Προσέγγιση παραγώγου (μαθηματικής) συνάρτησης σε σημείο x.

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

  Γράφουμε μια γενικευμένη συνάρτηση Python, με παράμετρο μια (οποιαδήποτε) προγραμματιστική συνάρτηση `f`, η οποία αντιστοιχεί σε μαθηματικές συναρτήσεις της μορφής $y = f(x)$.

```
def deriv(f, x, h=1e-8):
    return (f(x+h) - f(x)) / h
```

  την οποία μπορούμε να καλέσουμε για να υπολογίσουμε προσεγγίσεις παραγώγων συναρτήσεων f μιας μεταβλητής. Για κάθε (μαθηματική) συνάρτηση της οποίας θέλουμε να υπολογίσουμε την παράγωγο γράφουμε κατάλληλη συνάρτηση Python που υπολογίζει την τιμή της σε κάποιο x. Π.χ. για το πολυώνυμο

$$g(x) = 2x^3 - 3x^2 + 5x - 10$$

```
def g(x):
    return 2*x**3 - 3*x**2 + 5*x - 10
```

  Για να υπολογίσουμε την παράγωγο

  - της g στο $x = 1$ με $h = 10^{-4}$
  - και της $\sqrt{x}$ στο $x = 2$ με την default τιμή $h = 10^{-8}$:

```
import math
def main():
    dg = deriv(g, 1, 1e-4)
    dsqrt = deriv(math.sqrt, 4)
    print(dg, dsqrt)
```

  και παίρνουμε `5.000300019997184` και `0.24999997627617176`, αντί των πραγματικών τιμών 5 και 0.25 αντίστοιχα.

Τυχαιότητητα - Προσομοίωση Monte Carlo

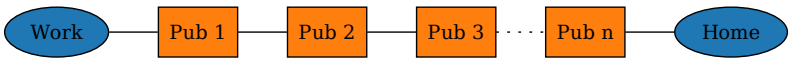
Η προσομοίωση Μόντε Κάρλο είναι μια τεχνική υπολογισμού που χρησιμοποιεί *τυχαία δείγματα (τυχαίους αριθμούς)* για να προσεγγίσει λύσεις σε προβλήματα που είναι δύσκολο να επιλυθούν με παραδοσιακές μαθηματικές μεθόδους.

Η βασική ιδέα πίσω από την προσομοίωση Μόντε Κάρλο είναι η χρήση τυχαίων δειγμάτων για να προσεγγίσουμε ένα πρόβλημα ή να υπολογίσουμε μια ποσότητα, όπως την πιθανότητα, την αναμενόμενη τιμή, ή κάποιον άλλον στατιστικό δείκτη.

1. **Δημιουργία Τυχαίων Δειγμάτων:** Στην προσομοίωση Μόντε Κάρλο, επιλέγουμε τυχαία δεδομένα από μια γνωστή κατανομή πιθανοτήτων. Για παράδειγμα, μπορούμε να ρίξουμε ένα ζάρι ή να δημιουργήσουμε τυχαίους αριθμούς από μια κατανομή όπως η κανονική κατανομή.
2. **Εκτέλεση Προβλέψεων ή Υπολογισμών:** Στη συνέχεια, με βάση τα τυχαία δείγματα, εκτελούμε υπολογισμούς ή προσομοιώνουμε τη διαδικασία που θέλουμε να αναλύσουμε. Αυτό μπορεί να περιλαμβάνει την προσομοίωση ενός τυχαίου παιχνιδιού, την εκτίμηση μιας αναμενόμενης τιμής, ή την ανάλυση ενός φυσικού συστήματος.
3. **Συνδυασμός Αποτελεσμάτων:** Τα αποτελέσματα *πολλών* τυχαίων προσομοιώσεων χρησιμοποιούνται για να υπολογίσουμε μια εκτίμηση για το ζητούμενο πρόβλημα. Ο τελικός υπολογισμός ή η εκτίμηση που προκύπτει από την προσομοίωση Μόντε Κάρλο βελτιώνεται όσο περισσότερες προσομοιώσεις εκτελούμε, καθώς η μέθοδος βασίζεται στη Νόμο των Μεγάλων Αριθμών. Όσο περισσότερες τυχαίες προσομοιώσεις εκτελούμε, τόσο καλύτερα προσεγγίζουμε την πραγματική τιμή που προσπαθούμε να υπολογίσουμε.

Ο περίπατος του μεθυσμένου

Είναι ένα πρόβλημα τυχαίου περιπάτου σε μια διάσταση. Ο κύριος Χ σχολάει από τη δουλειά του (κόμβος 0), στο παρακάτω γράφημα και βαδίζει προς το σπίτι του (κόμβος $n+1$). Στη διαδρομή υπάρχουν n pubs, και ο κ. Χ, που έχει μια ”αδυναμία” στη μπύρα, αποφασίζει να σταματήσει σε όποια pub συναντήσει στο δρόμο του και να πιει μια μπύρα. Έχοντας όμως ένα γενικότερο πρόβλημα προσανατολισμού, βγαίνοντας από κάθε pub, κινείται ισοπίθανα με τυχαίο τρόπο είτε δεξιά είτε αριστερά, με συνέπεια να επισκεφτεί κάποιες από τις pubs περισσότερες από μία φορά.



Θέλουμε να υπολογίσουμε ποιος είναι ο μέσος όρος από μπύρες που θα καταναλώσει μέχρι τελικά να φτάσει σπίτι του.

Αρχικά προσομοιώνουμε μια διαδρομή:

```
import random


def drunkards_walk(n):
    number_of_beers = 0
    x = 0
    while x != n+1:
        if x == 0:
            x += 1
        else:
            p = random.random()
            if p < 0.5:
                x -= 1
            else:
                x += 1
        if 1 <= x <= n:
            number_of_beers += 1
    return number_of_beers
```

Στη συνέχεια εκτελούμε nsims προσομοιώσεις για να υπολογίσουμε το ζητούμενο.

```
def main():
    nsims = int(input('Αριθμός προσομοιώσεων = '))
    npubs = int(input('Αριθμός pubs στη διαδρομή = '))
    total_beers = 0
    for _ in range(nsims):
        beers = drunkards_walk(npubs)
        total_beers += beers
    print(f'Μέσος όρος μπυρών = {total_beers/nsims}')
```

(Παρατηρήστε ένα ακόμα ιδίωμα της Python: το παραπάνω for χρησιμοποιείται αποκλειστικά και μόνο για να γίνει ένας αριθμός επαναλήψεων και δεν χρειάζεται κάπου η μεταβλητή-μετρητής. Στις περιπτώσεις αυτές μπορούμε να χρησιμοποιήσουμε ως σύμβαση τη μεταβλητή _ (κάτω παύλα) για να υποδηλώσουμε ότι η μεταβλητή δεν χρησιμοποιείται στην επανάληψη).

Οι τυχαίοι περίπατοι έχουν χρησιμοποιηθεί για τη μοντελοποίηση διαφορετικών φαινομένων σε πολλά πεδία μελέτης:

Πεδίο	Θέμα
Οικονομικά	τιμές μετοχών
Πληροφορική	ανάλυση δικτύων
Φυσική	κίνηση Brown
Γενετική Πληθυσμών	γενετική παρέκκλιση
Νευροεπιστήμη	ακολουθία πυροδότησης νευρώνων
Ψυχολογία	λήψη αποφάσεων
Αντίληψη	οφθαλμικές κινήσεις προσήλωσης

Μια από τις πιο ενδιαφέρουσες και σημαντικές εφαρμογές βρίσκεται στην καρδιά του PageRank, της μεθόδου που χρησιμοποιεί το Google για να καθορίσει τη σχετική σπουδαιότητα μιας ιστοσελίδας

που στηρίζεται στη βασική δομή συνδέσμων του Διαδικτύου. Στο παρασκήνιο "κρύβεται" ένας τυχαίος περίπατος όπου ένας "random web surfer" πηδάει από ιστοσελίδα σε ιστοσελίδα. Όταν ο surfer είναι σε μια ιστοσελίδα με k εξωτερικούς συνδέσμους, επιλέγει έναν από αυτούς στην τύχη και "πηγαίνει εκεί". Έτσι, αν υπάρχουν πέντε εξωτερικοί σύνδεσμοι και ένας από αυτούς αναφέρεται στην προσωπική σας ιστοσελίδα, τότε με πιθανότητα 0.2 η επόμενη στάση του surfer θα είναι εκεί. Αν δεν υπάρχουν εξωτερικοί σύνδεσμοι, τότε η επόμενη στάση του surfer καθορίζεται επιλέγοντας τυχαία μια από τις 10 δισεκατομμύρια περίπου ιστοσελίδες που αποτελούν σήμερα το Διαδίκτυο. Ο αλγόριθμος PageRank προσομοιώνει αυτή τη διαδικασία για ένα μεγάλο αριθμό από random web surfers και καταγράφοντας τις μετακινήσεις τους είναι σε θέση να κατατάξει όλες τις ιστοσελίδες ανάλογα με τη σημασία τους. Η κατάταξη αυτή καθορίζει εν μέρει την απόκριση της μηχανής αναζήτησης του Google όταν υποβάλουμε ένα ερώτημα.

Το πρόβλημα Monty Hall

Το πρόβλημα προέκυψε από ένα τηλεπαιχνίδι της αμερικάνικης τηλεόρασης:

Ένας διαγωνιζόμενος βρίσκεται μπροστά σε τρεις κλειστές πόρτες. Πίσω από μία από αυτές υπάρχει ένα έπαθλο), ενώ πίσω από τις υπόλοιπες δύο πόρτες υπάρχουν κατσίκες (οι αποτυχίες). Ο διαγωνιζόμενος καλείται να επιλέξει μία πόρτα, χωρίς να γνωρίζει τι υπάρχει πίσω από καμία από αυτές.

Αφού ο διαγωνιζόμενος κάνει την επιλογή του, ο παρουσιαστής (που γνωρίζει ποια πόρτα κρύβει το αυτοκίνητο) ανοίγει μία από τις άλλες δύο πόρτες, αποκαλύπτοντας πάντα μια κατσίκα. Στη συνέχεια, ο διαγωνιζόμενος έχει την ευκαιρία να αλλάξει την αρχική του επιλογή με την τρίτη, κλειστή πόρτα, ή να παραμείνει στην αρχική του επιλογή.

Η ερώτηση είναι: Ποια στρατηγική πρέπει να ακολουθήσει ο διαγωνιζόμενος για να μεγιστοποιήσει τις πιθανότητες να κερδίσει το αυτοκίνητο? Να αλλάξει πόρτα ή να μείνει στην αρχική του επιλογή?

Η πιθανότητα να κερδίσει κάποιος αν παραμείνει στην αρχική του επιλογή είναι 1/3 ενώ αν αλλάξει αυξάνεται στα 2/3 και ένας απλός τρόπος να το δείξουμε είναι με υπολογιστική προσομοίωση. Εκτελούμε `number_of_tests` προσομοιώσεις και καταγράφουμε πόσες φορές κέρδισε ο διαγωνιζόμενος όταν άλλαξε την αρχική του επιλογή (`switch`) και πόσες (`noswitch`) όταν παρέμεινε σε αυτήν.

```
from random import randint


def monty_hall(number_of_tests):
    switch = 0
    noswitch = 0
    for _ in range(number_of_tests):
        prize = randint(1, 3)          # Πόρτα με το βραβείο
        choice = randint(1, 3)         # Πόρτα που επιλέγει ο παίκτης
        if choice == prize:
            noswitch += 1
        else:
            switch += 1
    return switch, noswitch
```

Υπολογισμός πιθανοτήτων

Τζογαδόρος με αρχικό κεφάλαιο `stake` βάζει μια σειρά στοιχημάτων του 1 ευρώ. Στόχος του είναι να σταματήσει μόλις συγκεντρώσει κάποιο ποσό, έστω `goal`. Προφανώς το παιχνίδι σταματάει είτε όταν πετύχει το στόχο του είτε όταν χρεοκοπήσει.

- Ποιες είναι οι πιθανότητες να κερδίσει?
- Πόσα στοιχήματα θα χρειαστεί να βάλει μέχρι να κερδίσει/χάσει?

Θεωρούμε ότι ο παίκτης έχει 50% πιθανότητες να κερδίσει (π.χ. παίζει μια σειρά από παιχνίδια "κορώνα-γράμματα"). Από θεωρία πιθανοτήτων, η απάντηση στα ερωτήματα αυτά είναι γνωστή:

- πιθανότητα επιτυχίας $\pi =$ `stake/goal`
- απαιτούμενος αριθμός παιχνιδιών $\nu =$ `stake*(goal-stake)`

Μπορούμε να επαληθεύσουμε τα παραπάνω θεωρητικά αποτελέσματα με υπολογιστική προσομοίωση, εκτελώντας `nsims` πειράματα (προσομοιώσεις) ενός παιχνιδιού.

```python
import random


def gambler(stake, goal, nsims=10000):

    bets = 0
    wins = 0

    for _ in range(nsims):  # nsims πειράματα

        cash = stake  # Ξεκινάμε κάθε πείραμα με το αρχικό ποσό

        # Προσομοίωση ένος παιχνιδιού: μια σειρά από στοιχήματα
        # μέχρι να πετύχει τον στόχο του ή να χρεωκοπήσει.
        while 0 < cash < goal:
            # Προσομοίωση ενός στοιχήματος
            bets += 1
            if random.random() < 0.5:
                cash += 1  # κερδίζει 1 ευρώ
            else:
                cash -= 1  # χάνει 1 ευρώ
        if cash == goal:
            wins += 1

    print(f'Πιθανότητα να κερδίσει: {wins/nsims}')
    print(f'Μέσος όρος στοιχημάτων μέχρι να χάσει/κερδίσει: {bets/nsims}')
```

Μερικά ενδεικτικά συγκριτικά αποτελέσματα θεωρίας (Θ) και υπολογισμού (Υ) με 10000 πειράματα:

stake	goal	π (Θ)	π (Υ)	ν (Θ)	ν (Υ)
20	100	0.2	0.1938	1600	1588
1	100	0.01	0.0096	99	96