Primtall og effektivitet

Skrevet av: Martin Strand

Kurs: Python
Tema: Tekstbasert, Kryptografi
Fag: Matematikk, Programmering
Klassetrinn: 8.-10. klasse, Videregående skole

Introduksjon

I matematikktimene har du lært om primtall, altså tall som er slik at de bare kan deles av seg selv og 1. Nå skal vi se hvordan vi kan undersøke om et vilkårlig heltall er et primtall eller ikke.

Når vi skal kryptere data er vi ofte avhengige av å jobbe med store primtall, og vi vil få bruk for det i senere oppgaver. For å jobbe med primtall, må vi først ha en måte å finne dem på.

Steg 1: Rest etter divisjon

Den grunnleggende måten vi gjør dette på er å dele på andre tall. La oss si at vi vil sjekke tallet 15. Tallet 2 går ikke opp i 15, så da sier vi at vi får en *rest*. Åpne IDLE, men uten å lage en ny fil. Der det står `>>>` kan vi skrive inn kode for å se hva den gjør.

Sjekkliste

- [ ] Skriv `15//2` og trykk Enter. Du skal da få svaret 7. Det er fordi 2 går opp 7 ganger i 15
- [ ] Skriv `15 % 2` og trykk Enter. Da skal du få 1. Til sammen betyr dette at 15 delt på 2 blir 7, med 1 til rest.
- [ ] Prøv det samme med `15//6` og `15 % 6`. Hva betyr svarene du får ut?
- [ ] Skriv `15 % 3`. Da får du 0. Hvorfor?

Steg 2: Enkel prøvedivisjon

Over fikk vi 0 når vi regnet ut `15 % 3`. Det betyr at 3 går opp i 15, og altså er ikke 15 et primtall. Det kan vi utnytte for å sjekke om noe er et primtall. Vi kan dermed lage en funksjon som tar inn et tall `n` og sjekker om alle tall som er mindre enn `n` går opp i `n`.

Sjekkliste

- [ ] Lag en ny fil med følgende kode

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

print(is_prime(15))
print(is_prime(29))
```

- [ ] Kjør koden, og sjekk at den skriver ut `False` og `True`, i den rekkefølgen. Det betyr at 15 ikke er et primtall, men at 29 er det.

- [ ] Fra hovedvinduet til IDLE kan vi nå fortsette å teste koden vår med andre tall. Prøv for eksempel å skrive `is_prime(524287)`

- [ ] Prøv `is_prime(2147483647)`. Hvis det ser ut som at det ikke skjer noe, er det bare fordi tallet er så stort at det tar veldig lang tid å gå gjennom alle tall som er mindre enn det. Du kan avbryte testen ved å velge **Shell --> Interupt Execution**

Steg 3: Forbedret prøvedivisjon

Vi har laget et program som gir riktig svar, men som bruker veldig lang tid på å gjøre det. Da må vi se om vi kan øke hastigheten på koden vår, og her kan vi få hjelp av et enkelt matematisk argument. Si at et tall n er et produkt av to andre, slik at $n = pq$. Da må enten p eller q være mindre enn (eller lik) kvadratroten av n.

(Hvis du er interessert, her er en forklaring på hvorfor: Hvis begge er større, så får vi følgende utregning: $n = pq > \sqrt{n}\sqrt{n} = n$. Da har vi $n > n$, og det er umulig. Derfor må minst en av *p* og *q* være mindre eller lik.)

I Python kan vi regne ut kvadratrøtter ved å bruke funksjonen `sqrt` som finnes i `math`-biblioteket. Siden `sqrt` kan returnere et flyttall, bruker vi også funksjonen `ceil` fra samme bibliotek for å runde opp til nærmeste heltall.

Sjekkliste

- [ ] Endre koden din fra over slik at den ser slik ut:

```
from math import sqrt, ceil

def is_prime(n):
    for i in range(2, ceil(sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

print(is_prime(15))
print(is_prime(29))
```

- [ ] Prøv igjen med `is_prime(2147483647)`. Denne gangen skal svaret ha kommet nesten med én gang. Ved å bare endre litt på en eneste linje har vi fått kode som gjør den samme jobben mye raskere.

- [ ] Prøv å forstå hvorfor vi må ha `+ 1` i koden over. Hint: Ta bort `+ 1` og test koden din på tallet 25.

Steg 4: Flere forbedringer?

Vi kan ennå gjøre koden vår raskere ved å sjekke for 2, og deretter hoppe over alle partallene. For å gjøre det enkelt, kan vi legge til en tredje parameter til `for`-løkka vår, og som sier hvor mye den skal øke hver gang. Standardverdien er 1, men vi endrer den til 2.

Sjekkliste

- [ ] Endre koden din fra over slik at den ser slik ut:

```
from math import sqrt, ceil

def is_prime(n):
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, ceil(sqrt(n)) + 1, 2):
        if n % i == 0:
            return False
    return True

print(is_prime(15))
print(is_prime(29))
```

- [ ] Sjekk at koden fungerer med noen tall du selv velger.

Men, hvorfor skal vi stanse her? Vi kan jo også dele på 3 og 5 til å begynne med, og da trenger vi ikke sjekke noen av tallene over som selv er delelige på 3 eller 5. Og slik kan vi fortsette, helt til vi kun tester med tallene som selv er primtall. Og, vi vet jo allerede hvordan vi finner små primtall. Vi kan altså prøve oss fram med den følgende koden.

- [ ] Endre koden din til å sjekke om tallet vi tester med er et primtall før vi gjør prøvedivisjonen.

```
from math import sqrt, ceil

def is_prime(n):
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, ceil(sqrt(n)) + 1, 2):
        if is_prime(i):
            if n % i == 0:
                return False
    return True

print(is_prime(15))
print(is_prime(29))
```

- [ ] Test den nye koden på 2147483647. Da merker du kanskje at det nå tar litt *lenger* tid enn det gjorde før. Det er fordi vi nå har endt opp med å gjøre mange flere primtallstester enn vi gjorde før. Denne siste endringen var derfor en blindvei, men slik er det ofte i programmering: Ikke alle ideer er like gode.

- [ ] Vi prøver et kompromiss. Vi kan begynne med å lage en liste over små primtall som vi allerede vet om, og teste med dem først. Deretter kjører vi testen slik vi har gjort først. Endre koden til det følgende:

```
from math import sqrt, ceil

def is_prime(n):
    small_primes = [2, 3, 5, 7, 11, 13, 17, 19]
    if n in small_primes:
        return True
    for prime in small_primes:
        if n % prime == 0:
            return False
    for i in range(21, ceil(sqrt(n)) + 1, 2):
        if is_prime(i):
            if n % i == 0:
                return False
    return True

print(is_prime(15))
print(is_prime(29))
```

Denne siste koden er den raskeste vi har hatt hittil, og hvis man skal sjekke veldig mange tall på kort tid, gjør den siste forbedringen vår at det vil gå ganske raskt.

- [ ] Helt til sist skal vi legge til to veldig viktige linjer. Vi har til nå *antatt* at brukeren alltid vil sende et tall til funksjonen vår. Men, det kan vi aldri vite helt sikkert. Når vi jobber med programmering (og spesielt når det er kode som handler om sikkerhet!) må vi passe på at koden bare kan brukes til det den skal. Vi starter derfor funksjonen vår med å sjekke at vi faktisk har fått inn et positivt heltall. Endre koden din slik:

```
(...)
def is_prime(n):
    if not (isinstance(n, (int, long)) and n > 1):
        return False
    small_primes = [2, 3, 5, 7, 11, 13, 17, 19]
(...)
```

Bonus

Ikke overraskende kan vi fortsatt gjøre noen forbedringer, men her må du gjøre mye av tenkingen og programmeringen selv. Her skal vi erstatte listen av små primtall med en liste datamaskinen lager selv. Det gjør vi ved hjelp av en algoritme som kalles *Eratosthenes sil*. Begynn med å lage en lang liste av boolske verdier (`True` / `False`) opp til den verdien vi skal sjekke.

```
def eratosthenes(max):
    candidates = [True] * max
```

Vi bruker indekseringen av listen som tolkning av hvilket tall vi ser på. Start derfor med å sette `candidates[0] = False` og `candidates[1] = False` , siden 0 og 1 ikke regnes som primtall. Deretter er logikken som følger, og det er du selv som må oversette dette til Python-kode:

```
for hvert element i candidates, opp til kvadratroten av max:
    hvis elementet er True: (f.eks. 7)
        sett alle multipler av elementet til False (14, 21, 28, ...)
gå gjennom listen og lag en ny som består av tallene som fortsatt er Tr
ue
returner den nye listen
```

Når du er ferdig med å implementere rutinen din, kan du sette den inn i kodefilen vi laget over, og endre koden slik at den ser slik ut:

```
from math import sqrt, ceil

def eratosthenes(upper_limit):
    // din kode her

def is_prime(n, small_primes=[2, 3, 5]):
    if not (isinstance(n, (int, long)) and n > 1):
        return False
    if n in small_primes:
        return True
    for prime in small_primes:
        if n % prime == 0:
            return False
    start = small_primes[-1] + 2
    for i in range(start, ceil(sqrt(n)), 2):
        if is_prime(i):
            if n % i == 0:
                return False
    return True

upper_bound = 1000000
small_primes = eratosthenes(upper_bound)

print(is_prime(15, small_primes))
print(is_prime(29, small_primes))
```

Effekten av det vi har gjort nå er at vi har en liste med små primtall, og den kan vi gjenbruke så mange ganger vi vil. Dersom vi skal teste veldig mange tall, har vi derfor gjort en stor del av det arbeidet én gang for alle.