# Lesson 1: Week 10 - Notes 16 Three Special Perfect Squares

**Created:** 2024-05-27T01:25:44.263266+10:00

## Three special perfect squares

## Week 10 - Three special perfect squares

## three_special_perfect_squares_v1.py

## three_special_perfect_squares_v2.py

## three_special_perfect_squares_v3.py

## Three special perfect squares

# Three special perfect squares

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

[1]:
```
from math import sqrt
```

We aim to find all triples $(a, b, c)$ of natural numbers such that:

- $a < b < c$;
- $a$, $b$ and $c$ are perfect squares;
- 0 occurs in none of $a$, $b$ and $c$;
- every nonzero digit occurs exactly once in one of $a$, $b$ and $c$.

For instance $(25, 841, 7396)$ is a solution since $25 = 5^2$, $841 = 29^2$, $7396 = 86^2$, and we see that each of 1, 2, 3, 4, 5, 6, 7, 8 and 9 occurs once and only once in 25, 841 or 7396.

A bad strategy would be to generate natural numbers, check whether they are perfect squares, and in case they aren't, discard them. A much better strategy is to generate nothing but perfect squares.

- Since we need 3 perfect squares and use up each of the 9 nonzero digits once and only once, $c$ can consist of at most 7 digits, leaving one digit to $a$ and 1 digit to $b$. There are three 1-digit nonzero perfect squares: 1, 4 and 9. So a largest possible value for $c$ is obtained by setting $a$ to 1, $b$ to 4, and $c$ to 9876532.
- A largest possible value for $a$ is 997: otherwise, $b$ would consist of at least 3 digits and $c$ would consist of at least 4 digits, for a total of at least 10 digits.
- A largest possible value for $b$ is 9998: otherwise, $c$ would consist of at least 5 digits and $a$ takes at least one digit, for a total of at least 10 digits.

We could therefore look for solutions with the following code structure:

[2]:
```
for i in range(1, int(sqrt(997)) + 1):
    a = i ** 2
    for j in range(i + 1, int(sqrt(9998)) + 1):
        b = j ** 2
        for k in range(j + 1, int(sqrt(9876532)) + 1):
            c = k ** 2
            # Check whether (a, b, c) is a solution;
            # display it if it.
```

Let us see how many triples would be considered with the previous code structure:

[3]:
```
sum(1 for i in range(1, int(sqrt(997)) + 1)
      for j in range(i + 1, int(sqrt(9998)) + 1)
          for k in range(j + 1, int(sqrt(9876532)) + 1)
```

```
)
```

[3]: 7936372

There is no need to consider candidates for $b$ (and $c$) in case the code generates a value for $a$ that can be discarded because it contains at least one occurrence of 0 or at least two occurrences of the same digit; there is also no need to consider candidates for $c$ in case the code generates a value for $b$ that can be discarded because it contains at least one occurrence of 0 or some digit occurs at least twice in $a$ and $b$ considered together. So a better code structure is:

[4]:
```
for i in range(1, int(sqrt(997)) + 1):
    a = i ** 2
    # If a contains 0 or many occurrences of the same digit:
    #     continue
    for j in range(i + 1, int(sqrt(9998)) + 1):
        b = j ** 2
        # If b contains 0 or digits that occur in a or many occurrences
        # f the same digit:
        #     continue
        for k in range(j + 1, int(sqrt(9876532)) + 1):
            c = k ** 2
            # Check whether (a, b, c) is a solution;
            # display it if it is.
```

It seems that 0 plays a special role, but one can "pretend" that 0 is being generated in the first place, in which case we want to consider only candidates for $a$ which consist of nothing but distinct digits none of which belongs to the set $S_0$ of digits "seen" before (namely, $\{0\}$), and then consider only candidates for $b$ which consist of nothing but distinct digits none of which belongs to the set $S_1$ of digits seen before (namely, the union of $S_0$ with the set of digits that occur in $a$), and then consider only candidates for $c$ which consist of nothing but distinct digits none of which belongs to the set $S_2$ of digits seen before (namely, the union of $S_1$ with the set of digits that occur in $b$).

We will examine three techniques to keep track of the digits seen before, and to check that the candidate for $a$, $b$ or $c$ under consideration consists of nothing but distinct new digits.

With the first technique, we work with sets of digit characters rather than sets of digits, e.g., with `{'0'}` rather than `{0}`, with `{'0', '2', '4', '5', '7', '8'}` rather than `{0, 2, 4, 5, 7, 8}`. Having such a set $S$ of digit characters for the digits seen before (or "pretended" to be seen before in the case of 0), and considering a candidate $n$ for $a$, $b$ or $c$, we get a string $s$ from $n$, then a set $T$ of characters from $s$. Then all digits in $n$ are distinct and different to the digits seen before iff the number of digits in $n$ plus the number of (distinct) digits seen before, the latter being the cardinality of $S$, is equal to the cardinality of $S \cup T$. The cardinality of a set can be computed with the `len()` function. The number of digits in $n$ is nothing but the length of $s$, which can also be computed with the `len()` function. The union of two sets can be computed with the `|` operator:

[5]:
```
# Assume that 0, 2 and 5 are the digits seen before
S = {'0', '2', '5'}

# All digits in b are distinct and different to the digits seen before,
```

```python
# so all good
b = 784
str(b), set(str(b)), S | set(str(b))
len(S) + len(str(b)), len(S | set(str(b)))

# Not all digits in b are distinct, so not good
b = 676
str(b), set(str(b)), S | set(str(b))
len(S) + len(str(b)), len(S | set(str(b)))

# Some digits in b have been seen before, so not good
b = 729
str(b), set(str(b)), S | set(str(b))
len(S) + len(str(b)), len(S | set(str(b)))
```

[5]: ('784', {'4', '7', '8'}, {'0', '2', '4', '5', '7', '8'})

[5]: (6, 6)

[5]: ('676', {'6', '7'}, {'0', '2', '5', '6', '7'})

[5]: (6, 5)

[5]: ('729', {'2', '7', '9'}, {'0', '2', '5', '7', '9'})

[5]: (6, 5)

Rather than using the | operator, one can also call the `union` method, to be distinguished from the `update()` method: the latter does not return a new set but changes the set to which the method is applied. Also note that one can get the union of more than two sets:

[6]:
```python
S = {2, 4, 6}

S.union((3, 5, 7)), S
S.update((3, 5, 7)), S
S.union((10, 11), {20, 21, 22}, [30]), S
S.update((10, 11), {20, 21, 22}, [30]), S
{2, 4, 6} | {10, 11} | {20, 21, 22} | {30}
```

[6]: ({2, 3, 4, 5, 6, 7}, {2, 4, 6})

[6]: (None, {2, 3, 4, 5, 6, 7})

[6]: ({2, 3, 4, 5, 6, 7, 10, 11, 20, 21, 22, 30}, {2, 3, 4, 5, 6, 7})

[6]: (None, {2, 3, 4, 5, 6, 7, 10, 11, 20, 21, 22, 30})

[6]: {2, 4, 6, 10, 11, 20, 21, 22, 30}

The same distinction applies to intersection, difference, and symmetric difference, which for intersection and difference (but not symmetric difference), also accept many arguments, and which also have their own operators:

[7]:
```
S = {2, 3, 4, 5, 6, 7}

S.intersection((4, 5, 6, 7, 8, 9)), S
S.intersection_update((4, 5, 6, 7, 8, 9)), S
S.intersection((5, 6, 7, 8), {6, 7, 8, 9}, [7, 8, 10]), S
S.intersection_update((5, 6, 7, 8), {6, 7, 8, 9}, [7, 8, 10]), S
{2, 3, 4, 5, 6, 7} & {5, 6, 7, 8} & {6, 7, 8, 9} & {7, 8, 10}
```

[7]: ({4, 5, 6, 7}, {2, 3, 4, 5, 6, 7})

[7]: (None, {4, 5, 6, 7})

[7]: ({7}, {4, 5, 6, 7})

[7]: (None, {7})

[7]: {7}

[8]:
```
S = {2, 3, 4, 5, 6, 7}

S.difference((0, 1, 4, 5)), S
S.difference_update((0, 1, 4, 5)), S
S.difference((2, 8, 9), {6, 8, 9, 10}, [10, 11]), S
S.difference_update((2, 8, 9), {6, 8, 9, 10}, [10, 11]), S
{2, 3, 4, 5, 6, 7} - {2, 8, 9} - {6, 8, 9, 10} - {10, 11}
```

[8]: ({2, 3, 6, 7}, {2, 3, 4, 5, 6, 7})

[8]: (None, {2, 3, 6, 7})

[8]: ({3, 7}, {2, 3, 6, 7})

[8]: (None, {3, 7})

[8]: {3, 4, 5, 7}

[9]:
```
S = {2, 3, 4, 5, 6}

S.symmetric_difference((4, 5, 6, 7, 8)), S
S.symmetric_difference_update((4, 5, 6, 7, 8)), S
{2, 3, 4, 5, 6} ^ {4, 5, 6, 7, 8} ^ {2, 3, 8, 9} ^ {0, 1}
```

[9]: ({2, 3, 7, 8}, {2, 3, 4, 5, 6})

[9]: (None, {2, 3, 7, 8})

[9]: {0, 1, 7, 9}

Let us write a function `digits_if_ok_1()` with two parameters, `number`, meant to be assigned a natural number, and `digits_seen_before`, meant to be assigned a set of digit characters. The function is expected to return `None` if as characters, it is not the case that the digits in `number` are all distinct and different to those in `digits_seen_before`; otherwise, `digits_if_ok_1()` is expected to return a new set of digit characters that extends `digits_seen_before` with those in `number`:

[10]:
```
def digits_if_ok_1(number, digits_seen_before):
    number_str = str(number)
    digits_seen_now = digits_seen_before | set(number_str)
    if len(digits_seen_now) != len(digits_seen_before) + len(number_str):
        return
    return digits_seen_now

digits_if_ok_1(784, {'0', '2', '5'})
print(digits_if_ok_1(676, {'0', '2', '5'}))
print(digits_if_ok_1(729, {'0', '2', '5'}))
```

[10]: {'0', '2', '4', '5', '7', '8'}

```
None
None
```

We can now find out how many triples will be considered with the better code structure previously outlined:

[11]:
```
nb_of_triples = 0

for i in range(1, int(sqrt(997)) + 1):
    a = i ** 2
    a_digits_and_0 = digits_if_ok_1(a, {'0'})
    if not a_digits_and_0:
        continue
    for j in range(i + 1, int(sqrt(9998)) + 1):
        b = j ** 2
        a_b_digits_and_0 = digits_if_ok_1(b, a_digits_and_0)
        if not a_b_digits_and_0:
            continue
        for k in range(j + 1, int(sqrt(9876532)) + 1):
            nb_of_triples += 1

nb_of_triples
```

[11]: 552226

To complete the implementation, it suffices to also check whether the digits in $c$ are distinct and different to those seen before, and if that is the case, check that 10 digits have been seen altogether (since we included 0 to start with). We complement the code with a count for the number of solutions being discovered:

[12]:
```
nb_of_solutions = 0

for i in range(1, int(sqrt(997)) + 1):
    a = i ** 2
    a_digits_and_0 = digits_if_ok_1(a, {'0'})
    if not a_digits_and_0:
        continue
    for j in range(i + 1, int(sqrt(9998)) + 1):
        b = j ** 2
        a_b_digits_and_0 = digits_if_ok_1(b, a_digits_and_0)
        if not a_b_digits_and_0:
            continue
        for k in range(j + 1, int(sqrt(9876532)) + 1):
            c = k ** 2
            a_b_c_digits_and_0 = digits_if_ok_1(c, a_b_digits_and_0)
            if not a_b_c_digits_and_0 or len(a_b_c_digits_and_0) != 10:
                continue
            print(f'{a:7} {b:7} {c:7}')
            nb_of_solutions += 1

print('Altogether,', nb_of_solutions, 'solutions have been discovered.')
```

```
      1       4 3297856
      1       4 3857296
      1       4 5827396
      1       4 6385729
      1       4 8567329
      1       4 9572836
      1      49  872356
      1      64  537289
      1     256   73984
      1     625   73984
      4      16  537289
      4      25  139876
      4      25  391876
      4     289   15376
      9     324   15876
     16      25   73984
     16     784    5329
     25     784    1369
     25     784    1936
     25     841    7396
     36      81   74529
```

```
     36       81   79524
     36      729    5184
     81      324    7569
     81      576    3249
     81      729    4356
    361      529     784
Altogether, 27 solutions have been discovered.
```

With the second technique, we work with sets of digits. Having such a set $S$ of digits for the digits seen before (or "pretended" to be seen before in the case of 0), and considering a candidate $n$ for $a$, $b$ or $c$, we extract the rightmost digit $d$ in $n$, give up on that candidate in case $d$ belongs to $S$ but add $d$ to $S$ otherwise, and proceed this way for all remaining digits in $n$, if any. Computation of $n$ modulo 10 and integer division by 10 return the rightmost digit and the remaining digits in $n$, respectively. The function `digits_if_ok_2()` implements this technique:

[13]:
```
def digits_if_ok_2(number, seen_digits):
    while number:
        number, digit = divmod(number, 10)
        if digit in seen_digits:
            return
        seen_digits.add(digit)
    return seen_digits

digits_if_ok_2(784, {0, 2, 5})
print(digits_if_ok_2(676, {0, 2, 5}))
print(digits_if_ok_2(729, {0, 2, 5}))
```

[13]:
```
{0, 2, 4, 5, 7, 8}
```

```
None
None
```

Assume that we adjust the code that discovers and outputs all solutions with the second technique by just

- changing `a_digits_and_0 = digits_if_ok_1(a, {'0'})` to `a_digits_and_0 = digits_if_ok_2(a, {0})`, and
- changing the other two calls to `digits_if_ok_1()` to calls to `digits_if_ok_2()`.

We know that there are solutions for $a = 5^2 = 25$ together with both $b = 28^2 = 784$ and $b = 29^2 = 841$. This means that

- at some point, `a_digits_and_0` will evaluate to `{0, 2, 5}`,
- `a_b_digits_and_0 = digits_if_ok_2(784, a_digits_and_0)` will be executed, and later
- `a_b_digits_and_0 = digits_if_ok_2(841, a_digits_and_0)` will be executed.

The following code fragment demonstrates that we will not get the expected results, and why:

[14]:
```
a_digits_and_0 = {0, 2, 5}
a_b_digits_and_0 = digits_if_ok_2(784, a_digits_and_0)
print(a_b_digits_and_0)
```

```
a_digits_and_0
a_b_digits_and_0 = digits_if_ok_2(841, a_digits_and_0)
print(a_b_digits_and_0)
a_digits_and_0
```

```
{0, 2, 4, 5, 7, 8}
```

[14]: {0, 2, 4, 5, 7, 8}

```
None
```

[14]: {0, 1, 2, 4, 5, 7, 8}

In the following code fragment, we see:

- `f(n)` make `a` denote what `n` denotes, namely, the integer 1, represented somewhere in memory, before the statement `a = 10` makes `a` denote another integer, namely, 10, represented elsewhere in memory;
- `g(A)` make `A` denote what `S` denotes, namely, the set {2}, represented somewhere in memory, before the statement `A = {20}` makes `B` denote another set, namely, {20}, represented elsewhere in memory;
- `h(B)` make `B` denote what `L` denotes, namely, the list [3], represented somewhere in memory, before the statement `B = [30]` makes `B` denote another list, namely, [30], represented elsewhere in memory.

[15]:
```
n = 1
S = {2}
L = [3]

def f(a):
    print(a)
    a = 10

def g(A):
    print(A)
    A = {20}

def h(B):
    print(B)
    B = [30]

f(n); n
g(S); S
h(L); L
```

```
1
```

[15]: 1

```
{2}
```

[15]:
```
{2}
```

```
[3]
```

[15]:
```
[3]
```

In the following code fragment, we see:

- `g(A)` make `A` denote what `S` denotes, namely, the set $\{2\}$, represented somewhere in memory, before the statement `A.add(20)` changes that set to $\{2, 20\}$ by letting the representation of $\{2, 20\}$ replace the representation of $\{2\}$ in memory;
- `h(B)` make `B` denote what `L` denotes, namely, the list $[3]$, represented somewhere in memory, before the statement `B.append(30)` changes that list to $[3, 30]$ by letting the representation of $[3, 30]$ replace the representation of $[3]$ in memory.

[16]:
```
S = {2}
L = [3]

def g(A):
    print(A)
    A.add(20)

def h(B):
    print(B)
    B.append(30)

g(S); S
h(L); L
```

```
{2}
```

[16]:
```
{2, 20}
```

```
[3]
```

[16]:
```
[3, 30]
```

We infer that `digits_if_ok_2()` should receive as second argument not the set of digits seen before, but a copy of that set; the copy will be extended with the digits in the number passed as first argument in case those digits are all distinct and different to the digits seen before, leaving the first argument unmodified; `set()`, `list()`, `tuple()`, return a set, a list, a tuple, respectively, consisting of the elements that make up their arguments:

[17]:
```
S = {1, 2, 3}
L = [1, 2, 3]
T = 1, 2, 3
s = '123'
```

```
set(S), set(L), set(T), set(s)
list(S), list(L), list(T), list(s)
tuple(S), tuple(L), tuple(T), tuple(s)
```

[17]: ({1, 2, 3}, {1, 2, 3}, {1, 2, 3}, {'1', '2', '3'})

[17]: ([1, 2, 3], [1, 2, 3], [1, 2, 3], ['1', '2', '3'])

[17]: ((1, 2, 3), (1, 2, 3), (1, 2, 3), ('1', '2', '3'))

The problematic code fragment can then be amended as follows:

[18]:
```
a_digits_and_0 = {0, 2, 5}
a_b_digits_and_0 = digits_if_ok_2(784, set(a_digits_and_0))
print(a_b_digits_and_0)
a_digits_and_0
a_b_digits_and_0 = digits_if_ok_2(841, set(a_digits_and_0))
print(a_b_digits_and_0)
a_digits_and_0
```

```
{0, 2, 4, 5, 7, 8}
```

[18]: {0, 2, 5}

```
{0, 1, 2, 4, 5, 8}
```

[18]: {0, 2, 5}

The solution based on the second technique is then a simple modification of the implementation based on the first technique:

[19]:
```
nb_of_solutions = 0

for i in range(1, int(sqrt(997)) + 1):
    a = i ** 2
    a_digits_and_0 = digits_if_ok_2(a, {0})
    if not a_digits_and_0:
        continue
    for j in range(i + 1, int(sqrt(9998)) + 1):
        b = j ** 2
        a_b_digits_and_0 = digits_if_ok_2(b, set(a_digits_and_0))
        if not a_b_digits_and_0:
            continue
        for k in range(j + 1, int(sqrt(9876532)) + 1):
            c = k ** 2
            a_b_c_digits_and_0 = digits_if_ok_2(c, set(a_b_digits_and_0))
            if not a_b_c_digits_and_0 or len(a_b_c_digits_and_0) != 10:
```

```
                continue
            print(f'{a:7} {b:7} {c:7}')
            nb_of_solutions += 1

print('Altogether,', nb_of_solutions, 'solutions have been discovered.')
```

```
      1       4 3297856
      1       4 3857296
      1       4 5827396
      1       4 6385729
      1       4 8567329
      1       4 9572836
      1      49  872356
      1      64  537289
      1     256   73984
      1     625   73984
      4      16  537289
      4      25  139876
      4      25  391876
      4     289   15376
      9     324   15876
     16      25   73984
     16     784    5329
     25     784    1369
     25     784    1936
     25     841    7396
     36      81   74529
     36      81   79524
     36     729    5184
     81     324    7569
     81     576    3249
     81     729    4356
    361     529     784
Altogether, 27 solutions have been discovered.
```

With the third technique, we work with natural numbers smaller than $2^{10}$ to encode sets of digits: given $k < 10$ and $0 \leq n_0 < ... < n_k \leq 9$, we let $2^{n_0} + \cdots + 2^{n_k}$ encode the set $\{n_0, \ldots, n_k\}$. In other words, we let the positions of the bits set to 1 in the representation of a number $e$ smaller than $2^{10}$ as a 10-bit number encode the members of the set encoded by $e$, the rightmost position being position 0, the second rightmost position being position 1, etc. The illustration code below makes use of **set comprehensions**:

[20]:
```
e = 0
e, f'{e:010b}', {i for i in range(10) if f'{e:010b}'[9 - i] == '1'}

e = 2 ** 0 + 2 ** 3 + 2 ** 5 + 2 ** 8
e, f'{e:010b}', {i for i in range(10) if f'{e:010b}'[9 - i] == '1'}
```

```
e = 2 ** 3 + 2 ** 5 + 2 ** 9
e, f'{e:010b}', {i for i in range(10) if f'{e:010b}'[9 - i] == '1'}

e = 2 ** 10 - 1
e, f'{e:010b}', {i for i in range(10) if f'{e:010b}'[9 - i] == '1'}
```

[20]: (0, '0000000000', set())

[20]: (297, '0100101001', {0, 3, 5, 8})

[20]: (552, '1000101000', {3, 5, 9})

[20]: (1023, '1111111111', {0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

Having a number $e$ encoding the digits seen before (or "pretended" to be seen before in the case of 0), and considering a candidate $n$ for $a$, $b$ or $c$, we extract the rightmost digit $d$ in $n$, give up on that candidate in case $d$ is one of the digits encoded in $e$, but let a new number encode $d$ together with the digits encoded in $e$ otherwise, and proceed this way for all remaining digits in $n$, if any. Before we see a possible implementation of the third technique, let us see binary bitwise operations in action.

Dividing a number $n$ by $2^0$, $2^1$, $2^3$, $2^4$... can be seen as "pushing" $n$ written in binary by 0, 1, 2, 3, 4... positions to the right, which can also be achieved with the `>>` operator:

[21]:
```
n = 22

for k in range(5):
    print(k, f'{n // 2 ** k:2}', f'{n >> k:2}', f'{n >> k:010b}')
```

```
0 22 22 0000010110
1 11 11 0000001011
2  5  5 0000000101
3  2  2 0000000010
4  1  1 0000000001
```

Multiplying a number $n$ by $2^0$, $2^1$, $2^3$, $2^4$... can be seen as "pushing" $n$ written in binary by 0, 1, 2, 3, 4... positions to the left, "filling the gaps" with 0's, which can also be achieved with the `<<` operator:

[22]:
```
n = 22

for k in range(5):
    print(k, f'{n * 2 ** k:3}', f'{n << k:3}', f'{n << k:010b}')
```

```
0  22  22 0000010110
1  44  44 0000101100
2  88  88 0001011000
3 176 176 0010110000
4 352 352 0101100000
```

In particular, $2^0$, $2^1$, $2^3$, $2^4$... are also 1 pushed 0, 1, 2, 3, 4... positions to the left: these are the numbers that in binary, have a single 1 at position 0, 1, 2, 3, 4...:

[23]:
```
for k in range(5):
    print(k, f'{2 ** k:2}', f'{1 << k:2}', f'{1 << k:010b}')
```

```
0  1  1 0000000001
1  2  2 0000000010
2  4  4 0000000100
3  8  8 0000001000
4 16 16 0000010000
```

With bitwise or, `|`, we get a number with a bit of 1 at a given position iff at least one operand has a bit of 1 at that position:

[24]:
```
x = 0
y = 2 ** 0 + 2 ** 3 + 2 ** 4
print(f'{x:6}', f'{y:6}', f'{x | y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x | y:06b}')

x = 2 ** 0 + 2 ** 3 + 2 ** 4
y = 2 ** 6 - 1
print(f'{x:6}', f'{y:6}', f'{x | y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x | y:06b}')

x = 2 ** 0 + 2 ** 2
y = 2 ** 1 + 2 ** 5
print(f'{x:6}', f'{y:6}', f'{x | y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x | y:06b}')

x = 2 ** 0 + 2 ** 2 + 2 ** 4
y = 2 ** 3 + 2 ** 4 + 2 ** 5
print(f'{x:6}', f'{y:6}', f'{x | y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x | y:06b}')
```

```
     0     25     25
000000 011001 011001
    25     63     63
011001 111111 111111
     5     34     39
000101 100010 100111
    21     56     61
010101 111000 111101
```

With bitwise and, `&`, we get a number with a bit of 1 at a given position iff both operands have a bit of 1 at that position:

[25]:
```
x = 0
y = 2 ** 0 + 2 ** 3 + 2 ** 4
```

```
print(f'{x:6}', f'{y:6}', f'{x & y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x & y:06b}')

x = 2 ** 0 + 2 ** 3 + 2 ** 4
y = 2 ** 6 - 1
print(f'{x:6}', f'{y:6}', f'{x & y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x & y:06b}')

x = 2 ** 0 + 2 ** 2
y = 2 ** 1 + 2 ** 5
print(f'{x:6}', f'{y:6}', f'{x & y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x & y:06b}')

x = 2 ** 0 + 2 ** 2 + 2 ** 4
y = 2 ** 3 + 2 ** 4 + 2 ** 5
print(f'{x:6}', f'{y:6}', f'{x & y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x & y:06b}')
```

```
     0     25      0
000000 011001 000000
    25     63     25
011001 111111 011001
     5     34      0
000101 100010 000000
    21     56     16
010101 111000 010000
```

With bitwise xor, ^, we get a number with a bit of 1 at a given position iff exactly one operand has a bit of 1 at that position:

[26]:
```
x = 0
y = 2 ** 0 + 2 ** 3 + 2 ** 4
print(f'{x:6}', f'{y:6}', f'{x ^ y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x ^ y:06b}')

x = 2 ** 0 + 2 ** 3 + 2 ** 4
y = 2 ** 6 - 1
print(f'{x:6}', f'{y:6}', f'{x ^ y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x ^ y:06b}')

x = 2 ** 0 + 2 ** 2
y = 2 ** 1 + 2 ** 5
print(f'{x:6}', f'{y:6}', f'{x ^ y:6}')
print(f'{x:06b}', f'{y:06b}', f'{x ^ y:06b}')

x = 2 ** 0 + 2 ** 2 + 2 ** 4
y = 2 ** 3 + 2 ** 4 + 2 ** 5
print(f'{x:6}', f'{y:6}', f'{x ^ y:6}')
```

```python
print(f'{x:06b}', f'{y:06b}', f'{x ^ y:06b}')
```

```
     0     25     25
000000 011001 011001
    25     63     38
011001 111111 100110
     5     34     39
000101 100010 100111
    21     56     45
010101 111000 101101
```

So to check whether a digit $d$ is one of the digits encoded in $e$, and get a new number $f$ that encodes $d$ together with the digits encoded in $e$ in case $d$ is a new digit, it suffices to take the bitwise or of $e$ with 1 pushed $d$ positions to the left; either $d$ is not a new digit in which case this is $e$, or $d$ is a new digit in which case this is the desired $f$. The following code fragment illustrates with $e$ encoding the set $\{0, 2, 4\}$, and $d$ ranging all possible digits:

[27]:
```python
e = 2 ** 0 + 2 ** 2 + 2 ** 5
print(f'  {e:3}', ' ' * 10, f'{e:010b}')
for d in range(10):
    x = 1 << d
    y = e | 1 << d
    print(d, f'{x:3}', f'{x:010b}', f'{y:010b}', y == e)
```

```
   37            0000100101
0   1 0000000001 0000100101 True
1   2 0000000010 0000100111 False
2   4 0000000100 0000100101 True
3   8 0000001000 0000101101 False
4  16 0000010000 0000110101 False
5  32 0000100000 0000100101 True
6  64 0001000000 0001100101 False
7 128 0010000000 0010100101 False
8 256 0100000000 0100100101 False
9 512 1000000000 1000100101 False
```

Based on these considerations, we can implement the third technique in the form of the function `digits_if_ok_3()`:

[28]:
```python
def digits_if_ok_3(number, digits_seen_before):
    while number:
        number, digit = divmod(number, 10)
        digits_seen_now = digits_seen_before | 1 << digit
        if digits_seen_now == digits_seen_before:
            return
        digits_seen_before = digits_seen_now
    return digits_seen_before
```

```python
k = 2 ** 0 + 2 ** 2 + 2 ** 5
k, f'{k:010b}', {i for i in range(10) if f'{k:010b}'[9 - i] == '1'}

e = digits_if_ok_3(784, 2 ** 0 + 2 ** 2 + 2 ** 5)
e, f'{e:010b}', {i for i in range(10) if f'{e:010b}'[9 - i] == '1'}
digits_if_ok_3(676, 2 ** 2 + 2 ** 4)
digits_if_ok_3(729, 2 ** 2 + 2 ** 4)
```

[28]: (37, '0000100101', {0, 2, 5})

[28]: (437, '0110110101', {0, 2, 4, 5, 7, 8})

We can then adjust the code that discovers and outputs all solutions with the first technique by just

- changing `a_digits_and_0 = digits_if_ok_1(a, {'0'})` to `a_digits_and_0 = digits_if_ok_3(a, 1)` since 1 encodes 0 as a singleton set,
- changing the other two calls to `digits_if_ok_1()` to calls to `digits_if_ok_3()`, and
- eventually checking whether `a_b_c_digits_and_0` is is not `None` and evaluates to $2^{10} - 1$, which in base 2, reads as 10 consecutive 1's, therefore encoding all 10 digits:

[29]:
```python
nb_of_solutions = 0

for i in range(1, int(sqrt(997)) + 1):
    a = i ** 2
    a_digits_and_0 = digits_if_ok_3(a, 1)
    if not a_digits_and_0:
        continue
    for j in range(i + 1, int(sqrt(9998)) + 1):
        b = j ** 2
        a_b_digits_and_0 = digits_if_ok_3(b, a_digits_and_0)
        if not a_b_digits_and_0:
            continue
        for k in range(j + 1, int(sqrt(9876532)) + 1):
            c = k ** 2
            a_b_c_digits_and_0 = digits_if_ok_3(c, a_b_digits_and_0)
            if not a_b_c_digits_and_0 or a_b_c_digits_and_0 != 2 ** 10 - 1:
                continue
            print(f'{a:7} {b:7} {c:7}')
            nb_of_solutions += 1

print('Altogether,', nb_of_solutions, 'solutions have been discovered.')
```

```
      1       4 3297856
      1       4 3857296
      1       4 5827396
      1       4 6385729
      1       4 8567329
```

```
      1       4 9572836
      1      49  872356
      1      64  537289
      1     256   73984
      1     625   73984
      4      16  537289
      4      25  139876
      4      25  391876
      4     289   15376
      9     324   15876
     16      25   73984
     16     784    5329
     25     784    1369
     25     784    1936
     25     841    7396
     36      81   74529
     36      81   79524
     36     729    5184
     81     324    7569
     81     576    3249
     81     729    4356
    361     529     784
Altogether, 27 solutions have been discovered.
```