Memoisation is a specific type of caching that is used as a software optimization technique. It remembers the output of a function based on the input(and other parameters in some cases), and stores it for later use. As soon as we have a cached result, there is no need to re-run the memoized function for the same set of inputs. Instead, we can just fetch the cached result and return it right away.

This algorithm is often used when dealing with expensive code, which is generally code that takes a long time to run or uses a lot of memory.

In Python, we can implement memoisation as a decorator, which is a function that takes another function as an input and has a function as its output.

In [1]:
```
def memoise(func):
    cache = dict()

    def memoised_func(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result

    return memoised_func
```

The 'memoise' decorator takes a function and returns a wrapped version of the same function that implements the caching logic ('memoised_func'). In the implementation above, we use a dictionary as a way to store our cache, because it is faster to use a key to look up a value in Python using this type of container.

To summarise, we check if the parameters are already in the cache whenever the decorated function gets called. If so, the cached result is returned instead of being re-computed. If the result isn't in the cache, it will update the cache to save some time in the future. Therefore, we first compute the missing result, store it in the cache, and then return it to the caller.

Let's put our implementation to practice with a simple example function that calculates the square of a number:

In [2]:

```
import time, random

times = []
def squaring(num):
    return num ** 2

# Generating array of size 100 with random ints between 1 and 10 inclusive
array = [random.randint(1,10) for x in range(1000000)]

t1 = time.time()
for i in range(len(array)):
    squaring(array[i])
t2 = time.time()
times.append(t2-t1)
```

In [3]:

```
print(f'Total times of operation:\n - Without memoisation: {times[0]}')
```

```
Total times of operation:
 - Without memoisation: 0.28479886054992676
```

In [4]:

```
cache = dict()
def memoise_squaring(num):
    if num in cache:
        return cache[num]
    else:
        cache[num] = num ** 2
        return cache[num]


t1 = time.time()
for i in range(len(array)):
    memoise_squaring(array[i])
t2 = time.time()
times.append(t2-t1)
```

In [5]:
```
print(f'Here is our cache: {cache}\n')
print(f'Total times of operation:\n - Without memoisation: {times[0]}\n - With memoisation: {times[1]}')
```

```
Here is our cache: {3: 9, 9: 81, 8: 64, 6: 36, 4: 16, 10: 100, 2: 4, 1: 1, 5: 25, 7: 49}

Total times of operation:
 - Without memoisation: 0.28479886054992676
 - With memoisation: 0.2030010223388672
```

There is also a module called "functools" which has a method that allows us to use memoisation called 'lrucache()':

In [6]:
```
from functools import lru_cache

@lru_cache(maxsize=10000)
def square_me(num):
    return  num ** 2

t1 = time.time()
for i in range(len(array)):
    square_me(array[i])
t2 = time.time()
times.append(t2-t1)

print(f'Here is our cache information:\n {square_me.cache_info()}')
```

```
Here is our cache information:
 CacheInfo(hits=999990, misses=10, maxsize=10000, currsize=10)
```

In [7]:
```
print(f'Total times of operation:\n Without memoisation: {times[0]}\n With memoisation: {times[1]}\n Using lru_cac
```

```
Total times of operation:
 Without memoisation: 0.28479886054992676
 With memoisation: 0.2030010223388672
 Using lru_cache: 0.15029001235961914
```