EX1: In this exercice I am doing 10 trials and taking the average of the number of collision with the 4 sizes of hash tables. Hp,m = { ha,b | 1<= a < p, 0<= b < p}

ha,b(k)=((ak+b)mod p) mod m

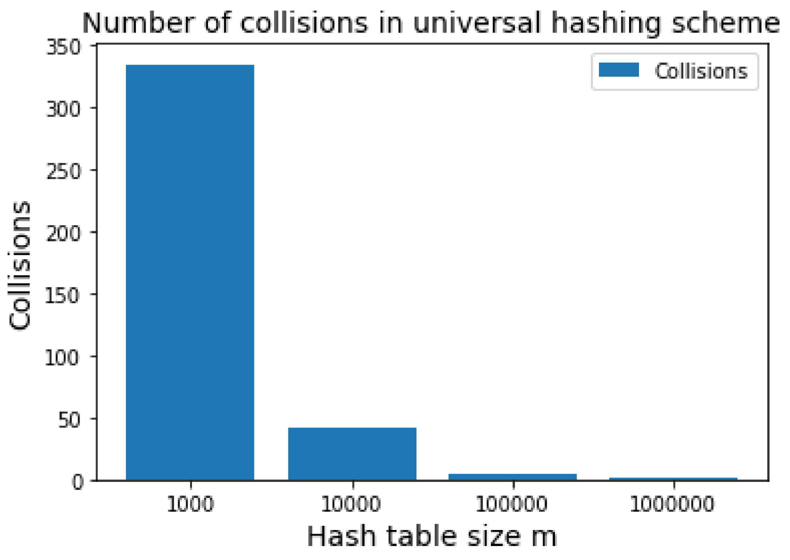
m – is a size of hash table

a and b randomly chosen numbers

p -is prime, should be greater than max number of keys and greater than m (p>m)

In [7]:
```
import random
import matplotlib.pyplot as plt
import statistics
key_array = []
file = open("1000_keys.txt", "rb")
for f in file:
    key_array.append(eval(f))
def check_collision(l, k, v):
    size_of_array = [1000, 10000, 100000, 1000000]
    n = 10
    to_plot = {}
    for m in size_of_array:
        for j in range(0,n):
            collisions = []
            collision = 0
            new_hash_array = []
            p = l
            a = k
            b = v
            for i in range(m):
                new_hash_array.append(0)

            for k in key_array:
                if new_hash_array[(((a * k + b) % p) % m)] == 0:
                    new_hash_array.insert((((a * k + b) % p) % m), k)
                else:
                    collision = collision + 1
            collisions.append(collision)
        to_plot[m] = statistics.mean(collisions)
    return to_plot
p = max(key_array)+20
to_plot = check_collision(p, random.randint(1,p), random.randint(0,p))
plt.title("Number of collisions in universal hashing scheme", fontsize=14)
plt.ylabel('Collisions', fontsize=14)
plt.xlabel('Hash table size m', fontsize=14)
plt.bar(range(len(to_plot)), to_plot.values(), align='center', label='Collisio
ns')
plt.xticks(range(len(to_plot)), to_plot.keys())
plt.legend()
plt.show()
```

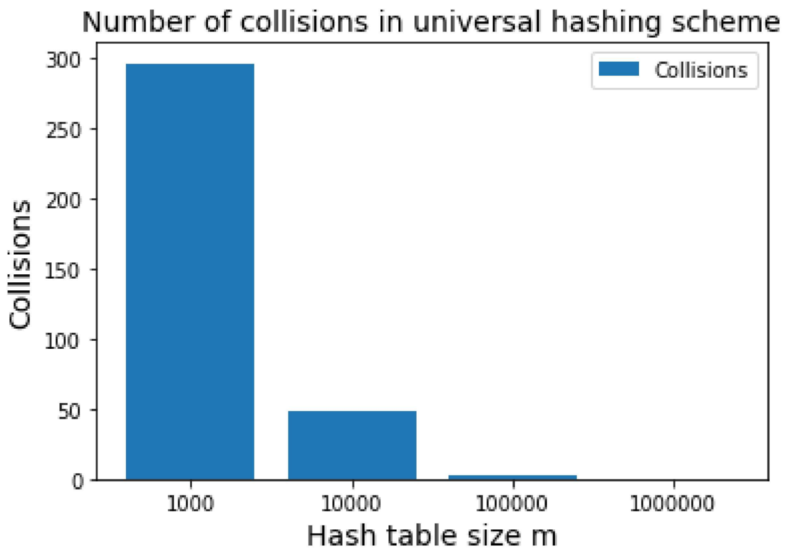


We can see that the table size makes a huge difference as the average number of collisions decreases exponentially, even with randomly chosen parameters.

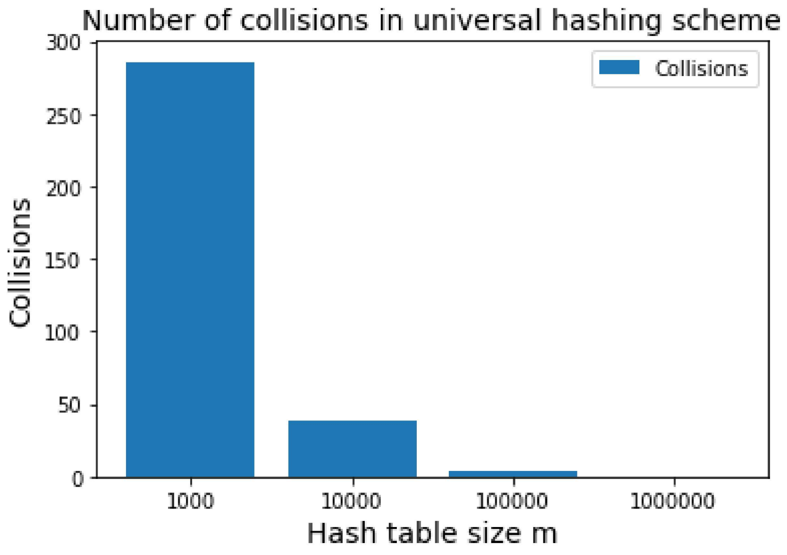
As we can obviously see from tables, by increasing size of table, the number or collision significantly decreased.

EX2: At first for checking I changed parameters of a , b.

In [2]:
```
p = max(key_array)+20
to_plot = check_collision(p, 98, random.randint(0,p))
plt.title("Number of collisions in universal hashing scheme", fontsize=14)
plt.ylabel('Collisions', fontsize=14)
plt.xlabel('Hash table size m', fontsize=14)
plt.bar(range(len(to_plot)), to_plot.values(), align='center', label='Collisio
ns')
plt.xticks(range(len(to_plot)), to_plot.keys())
plt.legend()
plt.show()
```



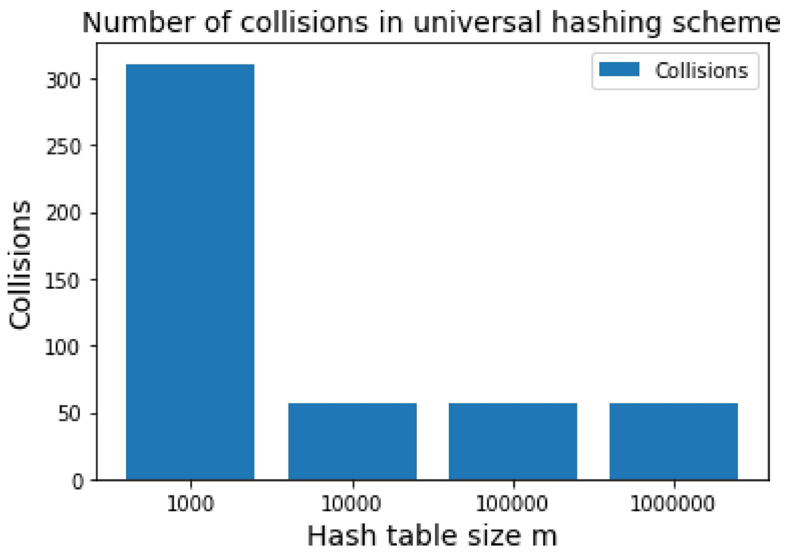
In [3]:
```
p = max(key_array)+20
to_plot = check_collision(p, random.randint(1,p), 86)
plt.title("Number of collisions in universal hashing scheme", fontsize=14)
plt.ylabel('Collisions', fontsize=14)
plt.xlabel('Hash table size m', fontsize=14)
plt.bar(range(len(to_plot)), to_plot.values(), align='center', label='Collisio
ns')
plt.xticks(range(len(to_plot)), to_plot.keys())
plt.legend()
plt.show()
```



As we can observe from the graphs, the number of collisions does not significantly change. Now in the code below, I changed the value of p. As we know, p prime should be bigger than m size. I changed number of p to less number than m (p<m) and got bad cases.

In [4]:
```
p = max(key_array)+20
to_plot = check_collision(9000, random.randint(1,p), random.randint(0,p))
print(to_plot)
plt.title("Number of collisions in universal hashing scheme", fontsize=14)
plt.ylabel('Collisions', fontsize=14)
plt.xlabel('Hash table size m', fontsize=14)
plt.bar(range(len(to_plot)), to_plot.values(), align='center', label='Collisio
ns')
plt.xticks(range(len(to_plot)), to_plot.keys())
plt.legend()
plt.show()
```

```
{1000: 311, 10000: 57, 100000: 57, 1000000: 57}
```



And we can observe it from the graph that of collisions increased.

EX3: A Bloom filter is a data structure designed to determine, rapidly and memory-efficiently, whether an element is present in a set. The base data structure of a Bloom filter is a Bit Vector .

The author deals with massive data (DNA sequences). Storing and processing such kind of data is memory and time consuming. He said that there are a lot of fragments and they have to be reassembled. He tells that we could find matching fragments and assemble the graph. To find out which fragment overlaps with another we have to look at each fragment, but in this case, it becomes memory intensive algorithm. To avoid this the author proposes to take a hash function and define Bloom Filter.

According to the definition of Bloom Filters, it is a probabilistic data structure that allows to store information about element which is belong to a set compactly. It could tell memory-efficiently if the element either definitely is not in the set or may be there.

This research represents a memory-efficient graph representation which can be used for analyzing the k-mer connectivity of metagenomic samples. Graph representation is based on Bloom filter, which allows storing assembly graphs in as little as 4 bits per k-mer efficiently.

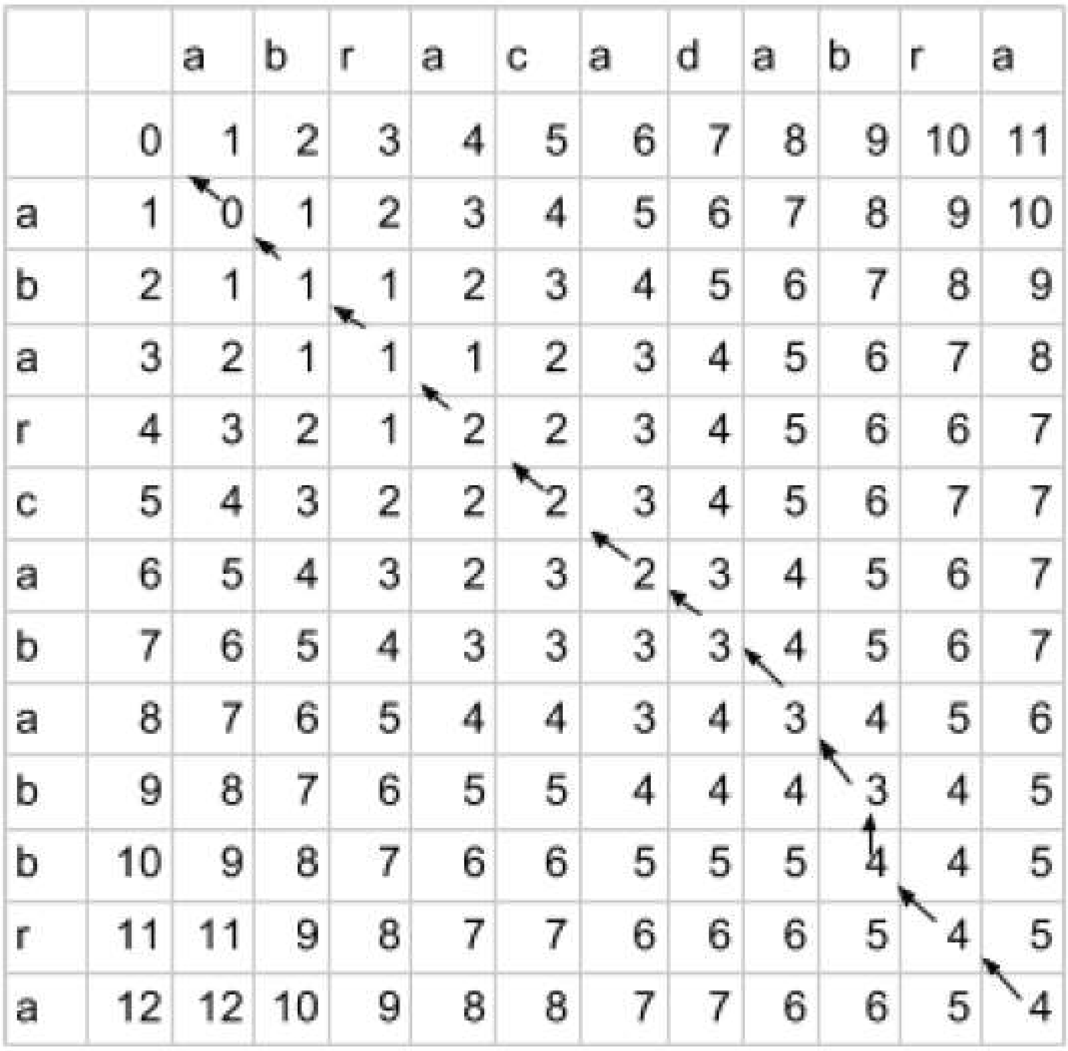
Author said that DNA sequence can’t directly assemble with Bloom filter, but the data structure could be used to grok graph properties and eliminate/break up data.

Researchers apply this data structure to the problem of partitioning assembly graphs into components as a prelude to assembly. Processing the data researchers noticed that if remove the repeated regions in the sequence assembler could work quicker.

EX4:

In [5]:
```
from IPython.display import Image
Image("4task.png")
```

Out[5]:



		a	b	r	a	c	a	d	a	b	r	a
	0	1	2	3	4	5	6	7	8	9	10	11
a	1	0	1	2	3	4	5	6	7	8	9	10
b	2	1	1	1	2	3	4	5	6	7	8	9
a	3	2	1	1	1	2	3	4	5	6	7	8
r	4	3	2	1	2	2	3	4	5	6	6	7
c	5	4	3	2	2	2	3	4	5	6	7	7
a	6	5	4	3	2	3	2	3	4	5	6	7
b	7	6	5	4	3	3	3	3	4	5	6	7
a	8	7	6	5	4	4	3	4	3	4	5	6
b	9	8	7	6	5	5	4	4	4	3	4	5
b	10	9	8	7	6	6	5	5	5	4	4	5
r	11	11	9	8	7	7	6	6	6	5	4	5
a	12	12	10	9	8	8	7	7	6	6	5	4

While filling the table we should look at whether the row and column elements match or not. After filling we can find the shortest path which should be near to diagonal. We should implement an insertion and 3 conversion operations to get the second string from the first. So the edit distance between 'abracadabra' and 'abarcababbra' is 4..

EX5: The knapsack problem or rucksack problem is a problem in combinatorial optimization : Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

https://www.researchgate.net/publication/224188966_A_Knapsack_problem_approach_for_achieving_efficient_en
(https://www.researchgate.net/publication/224188966_A_Knapsack_problem_approach_for_achieving_efficient_er

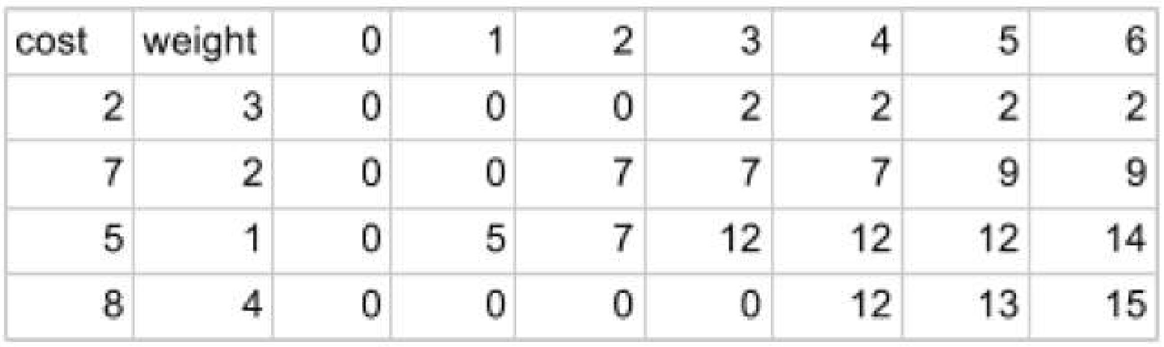
Interesting application of Knapsack problem: Home Energy Management. In order to achieve an efficient energy consumption level in the residential sector of a smart grid, the end-users are equipped with various smart home energy controller technologies. The devices are provided to inform the consumers about their consumption pattern by showing or sending different kinds of consumptional information to them. This kind of information is provided to assist them in making decisions about altering their consumption behaviour or to urge them to modify their life style during peak hours. We propose that the energy home controllers should offer preferred and optimal scenarios to support end-users when making a decision about their consumption. Effective scenarios should emerge from consumer's life style and preferences. In this paper, we will apply AHP methodology to quantify the consumer's preferences for using appliances during peak periods when the price has increased, and use the Knapsack problem approach to achieve the optimal solution for managing the appliances. With this approach, not only will the cost of electricity not escalate during peak hours, but also user preferences, satisfaction and minimum change to current life style will be considered.

Our constraint is 6 so we should fill the first raw from 0 up to 6. When we fill the cells with costs, we should compare the item weights with the given weights. If the item weights is bigger than the given weights we should put 0 to the cells appropriately. If the given weight is bigger than item weight(or the sum of items weights), we fill the cell with the item cost(or the the sum of items costs).

The max item cost is 15

```
In [6]: Image("5task.png")
```

Out[6]:



cost	weight	0	1	2	3	4	5	6
2	3	0	0	0	2	2	2	2
7	2	0	0	7	7	7	9	9
5	1	0	5	7	12	12	12	14
8	4	0	0	0	0	12	13	15