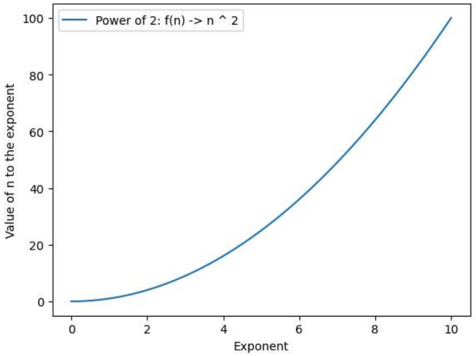
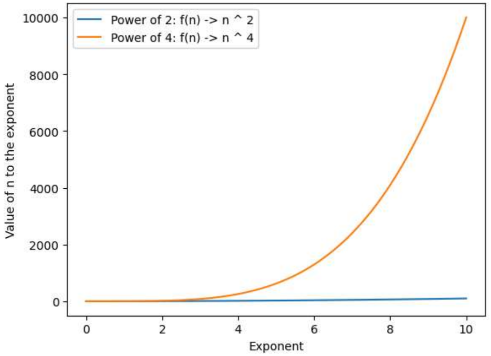
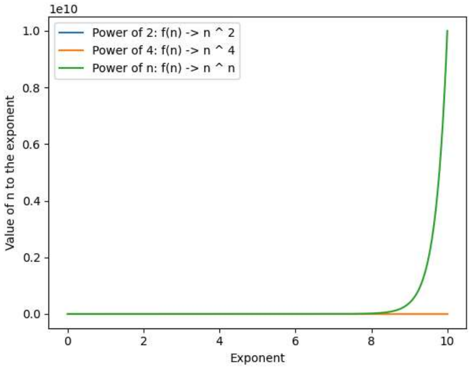
In [17]:
```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 1000)
y = x ** 2

_, ax = plt.subplots()
ax.plot(x, y, label="Power of 2: f(n) -> n ^ 2")

plt.xlabel("Exponent"), plt.ylabel("Value of n to the exponent")

plt.legend()
plt.show()
```



In [16]:
```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 1000)
y1, y2 = x ** 2, x ** 4

_, ax = plt.subplots()

ax.plot(x, y1, label="Power of 2: f(n) -> n ^ 2")
ax.plot(x, y2, label="Power of 4: f(n) -> n ^ 4")

plt.xlabel("Exponent"), plt.ylabel("Value of n to the exponent")

plt.legend()
plt.show()
```

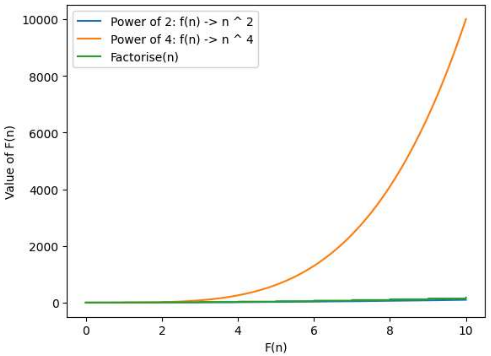


In [15]:
```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 1000)
y1, y2, y3 = x ** 2, x ** 4, np.array(list(map(lambda i: i ** i, x)))

_, ax = plt.subplots()

ax.plot(x, y1, label="Power of 2: f(n) -> n ^ 2")
ax.plot(x, y2, label="Power of 4: f(n) -> n ^ 4")
ax.plot(x, y3, label="Power of n: f(n) -> n ^ n")

plt.xlabel("Exponent"), plt.ylabel("Value of n to the exponent")

plt.legend()
plt.show()
```



In [26]:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 1000)
y1, y2 = x ** 2, x ** 4

_, ax = plt.subplots()
ax.plot(x, y1, label="Power of 2: f(n) -> n ^ 2")
ax.plot(x, y2, label="Power of 4: f(n) -> n ^ 4")


def algorithm(pn):
    n = pn
    l = 0  # counter for amount of iterations
    factors = []
    while n <= pn * 2:  # Do not go past MAX
        factor = 1  # 1 is a factor of any integer
        while factor <= n:  # Factors are <= the number
            if n % factor == 0:  # Is it a factor of n?
                pass
            factor += 1  # Try the next number
            l += 1
        l += 1
        n += 1
    return l


ax.plot(x, [algorithm(n) for n in x], label="Factorise(n)")

plt.xlabel("F(n)"), plt.ylabel("Value of F(n)")

plt.legend()
plt.show()
```



In [ ]:

```
# Import the Timer class defined in the module
from timeit import Timer

pop_zero = Timer("_.pop(0)", "from __main__ import _")
pop_end = Timer("_.pop()", "from __main__ import _")
print("pop(0) pop()")
pzs, pts = [], []

x = range(1000000, 100000001, 1000000)

l = []

for i in x:
    _ = list(range(i))
    pt = pop_end.timeit(number=100)
    _ = list(range(i))
    pz = pop_zero.timeit(number=100)
    print("%15.5f, %15.5f" % (pz, pt))
    pzs.append(pz), pts.append(pt)

import matplotlib.pyplot as plt

print(list(x))
```

```python
_, ax = plt.subplots()

ax.scatter(list(x), pzs, label="Time for 100 iterations of pop 0")
ax.scatter(list(x), pts, label="Time for 100 iterations of pop end")

plt.xlabel("Time"), plt.ylabel("List size")

plt.legend()
plt.show()
```

In [ ]:
```python
import timeit

x = range(10000, 1000001, 20000)

ls, ds = [], []

for i in x:
    t = timeit.Timer("random.randrange(%d) in _" % i, "from __main__ import random, _")
    _ = list(range(i))
    lst_time = t.timeit(number=100)
    _ = {j: None for j in range(i)}
    d_time = t.timeit(number=100)
    ls.append(lst_time), ds.append(d_time)

    print("%d,%10.3f,%10.3f" % (i, lst_time, d_time))

import matplotlib.pyplot as plt

_, ax = plt.subplots()

ax.plot(list(x), ls, label="Time for 100 iterations of checking for i in list")
ax.plot(list(x), ds, label="Time for 100 iterations of checking for i in dictionary")

plt.xlabel("Time"), plt.ylabel("Container size")

plt.legend()
plt.show()
```

In [ ]:
```python
### 3.5.1 What does “Amortised Worst Case” mean? [2 marks]
# The basic idea is that a worst-case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost. [Source](https://en.wikipedia.org/wiki/Amortized_analysis#Method)
### 3.5.2 How is it different to the “Average Case”? [2 marks]
# Amortised worst case focuses on accounting for occasional worst case operations whereas average case focuses only on a typical operation
### 3.5.3 What is a “deque”? [1 marks]
# A deque is a double ended queue [Source](https://en.wikipedia.org/wiki/Double-ended_queue#:~:text=In%20computer%20science%2C%20a%20double,)%20or%20back%20(tail).)
### 3.5.4 Create a 7 x 4 table that contains four columns for the four data structures listed on the page: List; deque; set; dict. Also create a row for each of the following seven operations: copy; membership (i.e. “x in s”); insert; delete; get item; set item; pop. Populate the table using the Average Case information from the webpage. [5 marks]

table = [
    # list
    ["n", "n", "n", "n", 1, 1],
    # deque
    ["n", None, None, None, None, None],
    # set
    [None, 1, None, None, None, None],
    # dict
    ["n", 1, None, 1, 1, 1]
]

### 3.5.5 What have you found out when creating this table? [2 marks]
# Some data structures may provide better time complexity but may not support entire suites of operations
```

In [53]:
```python
import numpy as np

np_arr = np.array(range(0, 18, 3))

arr1 = np.fromfunction(lambda a, b: a ** b, (3, 3))
arr2 = np.fromiter((i ** 2 for i in range(9)), int)
arr3 = np_arr[[0, 2, 4]]
arr4 = np.logspace(1, 3, num=5)

print(arr1, arr2, arr3, arr4, sep="\n")
```

```
[[1. 0. 0.]
 [1. 1. 1.]
 [1. 2. 4.]]
[ 0  1  4  9 16 25 36 49 64]
[ 0  6 12]
[  10.          31.6227766   100.          316.22776602 1000.        ]
```

In [60]:
```python
from numpy import median, std

print(median(range(1, 10_000)), std(range(10_000)))
```

```
5000.0 2886.751331514372
```

In [61]:
```python
# 3.8 onwards were not completed
```