Random list generater

generate_random_number_list(n) fundtion generates N random numbers between 0 -- 1,000,000,000

In [1]:

```
import random

def generate_random_number_list(n):
    lst = []
    for i in range(n):
        lst.append(random.randrange(0, 1000000000))
    return lst
```

In [2]:

```
# example
generate_random_number_list(10)
```

Out[2]:

```
[775013742,
 838899964,
 467659761,
 964752423,
 73284383,
 837544825,
 702083919,
 42760325,
 352321044,
 272440831]
```

find_max algorithm analysis

Check p23 in the textbook, or slide #39 in the lecture 2.

In [3]:

```
# Original function definition
def find_max(data):
    # it returns the maximum
    biggest = data[0]
    for val in data:
        if val > biggest:
            biggest = val
    return biggest
```

Experimental Study

In this section, we evaluate find_max experimentally, by varing the data size.

In [4]:

```
# Let's first import a timer
from time import time
```

In [ ]:

```
# Evaluate find_max for varying data sizes
# Note: this will take some time

data_sizes = []
execution_time = []
for data_size in range(1000, 100000, 1000):
    data = generate_random_number_list(data_size)

    start_time = time()
    find_max(data)
    end_time = time()
    elapsed = end_time - start_time

    data_sizes.append(data_size)
    execution_time.append(elapsed)

print('DONE!')
```

In [ ]:

```
# Plot the execution times on different data sizes

import matplotlib.pyplot as plt

fig = plt.figure()
ax=fig.add_axes([0,0,1,1])
ax.scatter(data_sizes, execution_time)
ax.set_xlabel('Data size')
ax.set_ylabel('Execution Time')
ax.set_title('find_max')
plt.show()
```

Asymptotic Analysis

In [ ]:

```
# Counting the number of operations in find_max
def find_max_counting(data):
    counter = [0, 0, 0]   # This list will store the number of operations, initialized to zeros

    biggest = data[0]           #
    counter[0]+=1               # OP 1 (assignment)

    for val in data:            #
        counter[1]+=1           # OP 2 (loop over the data)

        if val > biggest:       # This if statement is evaluated the same times to OP 2

            biggest = val       #
            counter[2]+=1       # OP 3 (assignment inside a loop)

    return biggest, counter
```

In [ ]:

```
# Evaluate find_max_counting for varying data sizes
# Note: this will take some time

data_sizes = []
OP1 = []
OP2 = []
OP3 = []

for data_size in range(1000, 100000, 1000):
    data = generate_random_number_list(data_size)

    result = find_max_counting(data)
    counted_ops = result[1]

    data_sizes.append(data_size)
    OP1.append(counted_ops[0])
    OP2.append(counted_ops[1])
    OP3.append(counted_ops[2])

    print(f'{data_size}\t{counted_ops}')

print('DONE!')
```

In [ ]:

```
# Plot the # of operations on different data sizes

import matplotlib.pyplot as plt

fig = plt.figure()
ax=fig.add_axes([0,0,1,1])
ax.scatter(data_sizes, OP1, label='OP1')
ax.scatter(data_sizes, OP2, label='OP2')
ax.scatter(data_sizes, OP3, label='OP3')
ax.set_xlabel('Data size')
ax.set_ylabel('# of operations')
ax.set_title('find_max Asymptotic analysis')
plt.legend()
plt.show()
```

Further Analysis of the algorithm

A more interesting question about **find_max** is how many times we might update the current “biggest” value.

In the worst case, if the data is given to us in increasing order, the biggest value is reassigned $n - 1$ times.

But what if the input is given to us in random order, with all orders equally likely; what would be the expected number of times we update the biggest value in this case?

To answer this question, note that we update the current biggest in an iteration of the loop only if the current element is bigger than all the elements that precede it. If the sequence is given to us in random order, the probability that the j^{th} element is the largest of the first j elements is $1/j$ (assuming uniqueness).

Hence, the expected number of times we update the biggest (including initialization) is

$$H_n = \sum_{j=1}^{n} 1/j$$

which is known as the n^{th} Harmonic number. It turns out that H_n is $O(\log n)$.

Therefore, the expected number of times the biggest value is updated by find max on a randomly ordered sequence is $O(\log n)$.

Finally, the execution operation analysis will be result in:

- OP1: Constant = 1
- OP2: Linear = n
- OP3: Harmonic = $\log n$
- Total = $1 + n + \log n$ = $O(n)$

In [ ]:

```

```