write a program that uses multiple threads to solve the problem of finding the integer in the range 1 to 10000 that has the largest number of divisors, but for the range 1 to 100000. At the end of the program, output the elapsed time, the integer that has the largest number of divisors, and the number of divisors that it has.

```
import threading
import time
def find_max_divisors(start, end, result):
  max_divisors = 0
  integer_with_max_divisors = 0
  for integer in range(start, end+1):
    divisors = 0
    for i in range(1, integer+1):
      if integer % i == 0:
       divisors += 1
      if divisors > max_divisors:
        max_divisors = divisors
        integer_with_max_divisors = integer
  result.append((max_divisors, integer_with_max_divisors))
def main():
  start_time = time.time()
  threads = []
  result = []
  for i in range(10):
    t = threading.Thread(target=find_max_divisors, args=(i*1000+1, (i+1)*1000,re
    threads.append(t)
    t.start()
  for t in threads:
    t.join()
  max_divisors, integer_with_max_divisors = max(result)
  elapsed_time = time.time() - start_time
  print("Elapsed time: {:.5f} seconds".format(elapsed_time))
  print("Integer with max divisors: {} ({}divisors)".format(integer_with_max_div
if __name__ == '__main__':
   main()
```

```
Elapsed time: 6.10658 seconds
Integer with max divisors: 9240 (64divisors)
```

We need to write a function called frequency. It takes a slice of strings and a worker count as parameters. The return value is a hashmap. The keys are all the letters those strings contain, their value the amount of times that letter appears. This needs to be done in worker_count number of process.

```
import multiprocessing

def frequency(strings, worker_count):
    manager = multiprocessing.Manager()
    return_dict = manager.dict()
    jobs = []

    for i in range(worker_count):
        p = multiprocessing.Process(target=count_letters, args=(strings[i::worke
        jobs.append(p)
        p.start()

    for job in jobs:
        job.join()

    result_dict = {}
    for letter in return_dict:
        result_dict[letter] = sum(return_dict[letter])

    return result_dict

def count_letters(strings, return_dict):
    for s in strings:
        for c in s:
            if c not in return_dict:
                return_dict[c] = multiprocessing.Array('i', [0]*multiprocessing.
            return_dict[c][multiprocessing.current_process()._identity[0]-1] +=
```

Implement three classes: Storage, Counter, and Printer. The Storage class should store an integer. The Counter class should create a thread that starts counting from 0 (0, 1, 2, 3 ...) and stores each value in the Storage class. The Printer class should create a thread that keeps reading the value in the Storage class and printing it. Write a program that creates an instance of the Storage class and sets up a Counter and a Printer object to operate on it.

```
import threading
import time
class Storage:
  def __init__(self):
    self.value = 0
  def set_value(self, value):
    self.value = value
  def get_value(self):
    return self.value
class Counter:
  def __init__(self, storage):
    self.storage = storage
    self.thread = threading.Thread(target=self.run)
    self.running = False
  def start(self):
    self.running = True
    self.thread.start()
  def stop(self):
    self.running = False
  def run(self):
    while self.running:
      self.storage.set_value(self.storage.get_value() + 1)
      time.sleep(1)
class Printer:
  def __init__(self, storage):
    self.storage = storage
    self.thread = threading.Thread(target=self.run)
    self.running = False
  def start(self):
    self.running = True
    self.thread.start()
  def stop(self):
    self.running = False
  def run(self):
    while self.running:
      print(self.storage.get_value())
      time.sleep(1)
storage = Storage()
counter = Counter(storage)
printer = Printer(storage)
counter.start()
printer.start()
time.sleep(10)
counter.stop()
printer.stop()
```

```
1
2
3
4
5
6
7
8
9
10
```

Write a program using multi-threads in which create a global integer vector. function FibonacciNumber(int, int) that takes 02 integers to calculate the next number in Fibonacci series and store it in the vector. In main create 02 threads and call assign the FinbonacciNumber function to each of the thread to calculate the next number and store the result in the integer vector. another thread that prints the Fibonacci series in parallel. FibonacciNumber(int n1, int n2); //function to calculate the next number in Fibonacci series. printFibSeries() const; //function to print the current status of series.

```
import threading
fibonacci_vector = [0, 1]
class FibonacciThread(threading.Thread):
  def __init__(self, n1, n2):
    super().__init__()
    self.n1 = n1
    self.n2 = n2
  def run(self):
    global fibonacci_vector
    next_fibonacci_number = self.n1 + self.n2
    fibonacci_vector.append(next_fibonacci_number)
class PrintFibonacciThread(threading.Thread):
  def run(self):
    global fibonacci_vector
    while True:
      print(fibonacci_vector)
      threading.Event().wait(1)
fibonacci_thread1 = FibonacciThread(0, 1)
fibonacci_thread2 = FibonacciThread(1, 1)
print_thread = PrintFibonacciThread()
fibonacci_thread1.start()
fibonacci_thread2.start()
print_thread.start()
fibonacci_thread1.join()
fibonacci_thread2.join()
print_thread.join()
```

Write a multi processing program in which the first process prints odd numbers between 100 to 200. Second Process print the prime number. Between 200 to 300. And third process generate Armstrong number between 100 to 300.

```
import multiprocessing
import math
def print_odd():
  for i in range(100, 201):
    if i % 2 == 1:
      print("Odd number:", i)
def print_prime():
  for i in range(200, 301):
    if i > 1:
      is_prime = True
  for j in range(2, int(math.sqrt(i)) + 1):
    if i % j == 0:
      is_prime = False
      break
    if is_prime:
      print("Prime number:", i)
def is_armstrong(number):
  sum = 0
  n = len(str(number))
  for digit in str(number):
    sum += int(digit) ** n
    return sum == number
def print_armstrong():
  for i in range(100, 301):
    if is_armstrong(i):
      print("Armstrong number:", i)
if __name__ == '__main__':
  p1 = multiprocessing.Process(target=print_odd)
  p2 = multiprocessing.Process(target=print_prime)
  p3 = multiprocessing.Process(target=print_armstrong)
  p1.start()
  p2.start()
  p3.start()
  p1.join()
  p2.join()
  p3.join()
```

```
Odd number: 101
Odd number: 103
Odd number: 105
Odd number: 107
Odd number: 109
Odd number: 111
Odd number: 113
Odd number: 115
Odd number: 117
Odd number: 119
Odd number: 121
Odd number: 123
Odd number: 125
Odd number: 127
Odd number: 129
Odd number: 131
Odd number: 133
Odd number: 135
Odd number: 137
Odd number: 139
Odd number: 141
Odd number:  Odd number:145

147Odd number:
Odd number:149143
Odd number:   151
Odd number: 153
Odd number: 155
Odd number: 157
Odd number: 159
Odd number: 161
Odd number: 163
Odd number: 165
Odd number: 167
Odd number: 169
Odd number: 171
Odd number: 173
Odd number: 175
Odd number: 177
Odd number: 179
Odd number: 181
Odd number: 183
Odd number: 185
Odd number: 187
Odd number: 189
Odd number: 191
Odd number: 193
Odd number: 195
Odd number: [0, 1, 1, 2]
[0, 1, 1, 2]
197
Odd number: 199
```

Write a Python program to define a subclass using threading and instantiate the subclass and trigger the thread which implement the task of building dictionary for each character from the the given input string.

```
import threading
class CharDictThread(threading.Thread):
  def __init__(self, string):
    threading.Thread.__init__(self)
    self.string = string
    self.char_dict = {}
  def run(self):
    for char in self.string:
      if char in self.char_dict:
        self.char_dict[char] += 1
      else:
        self.char_dict[char] = 1

input_string = "Ronit Kumar"
char_dict_thread = CharDictThread(input_string)
char_dict_thread.start()
char_dict_thread.join()
print(char_dict_thread.char_dict)
```

```
    {'R': 1, 'o': 1, 'n': 1, 'i': 1, 't': 1, ' ': 1, 'K': 1, 'u': 1, 'm': 1, 'a
```

Write a program Find that searches all files specified on the command line and prints out all lines containing a reserved word. Start a new thread for each file.

```
import threading
class FileSearchThread(threading.Thread):
  def __init__(self, file_name, search_word):
    threading.Thread.__init__(self)
    self.file_name = file_name
    self.search_word = search_word
  def run(self):
    with open(self.file_name) as f:
      for line in f:
        if self.search_word in line:
          print(f"Found '{self.search_word}' in file '{self.file_name}':{line.rs
file_name = "app.txt"
search_word = "Advanced"
threads = []
thread = FileSearchThread(file_name, search_word)
threads.append(thread)
thread.start()
# Wait for all threads to complete
for thread in threads:
  thread.join()
```

```
Exception in thread Thread-35:
Traceback (most recent call last):
  File "/usr/lib/python3.9/threading.py", line 980, in _bootstrap_inner
    self.run()
  File "<ipython-input-14-bb032ffeb460>", line 8, in run
FileNotFoundError: [Errno 2] No such file or directory: 'app.txt'
```

Implement the merge sort algorithm by spawning a new thread for each smaller MergeSorter. Hint: Use the join method of the Thread class to wait for the spawned threads to finish.

```
import threading
def merge_sort(arr):
  if len(arr) <= 1:
    return arr
  mid = len(arr) // 2
  left = merge_sort(arr[:mid])
  right = merge_sort(arr[mid:])
  return merge(left, right)
def merge(left, right):
  result = []
  i = j = 0
  while i < len(left) and j < len(right):
    if left[i] < right[j]:
      result.append(left[i])
      i += 1
    else:
      result.append(right[j])
      j += 1
    result += left[i:]
    result += right[j:]
    return result
class MergeSorter(threading.Thread):
  def __init__(self, arr):
    threading.Thread.__init__(self)
    self.arr = arr
  def run(self):
    self.arr = merge_sort(self.arr)
def multithreaded_merge_sort(arr, num_threads):
    if num_threads <= 1:
      return merge_sort(arr)
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    left_sorter = MergeSorter(left)
    right_sorter = MergeSorter(right)
    left_sorter.start()
    right_sorter.start()
    left_sorter.join()
    right_sorter.join()
    return merge(left_sorter.arr, right_sorter.arr)
if __name__ == '__main__':
    arr = [38, 27, 43, 3, 9, 82, 10]
    print(multithreaded_merge_sort(arr, 2))
```

```
[3, 27, 38, 43, 9, 10, 82]
```