Core 3

May 1, 2023

1 Core 3 Results

[1]:
```
import numpy as np, matplotlib.pyplot as plt, time, core_three as c3
```

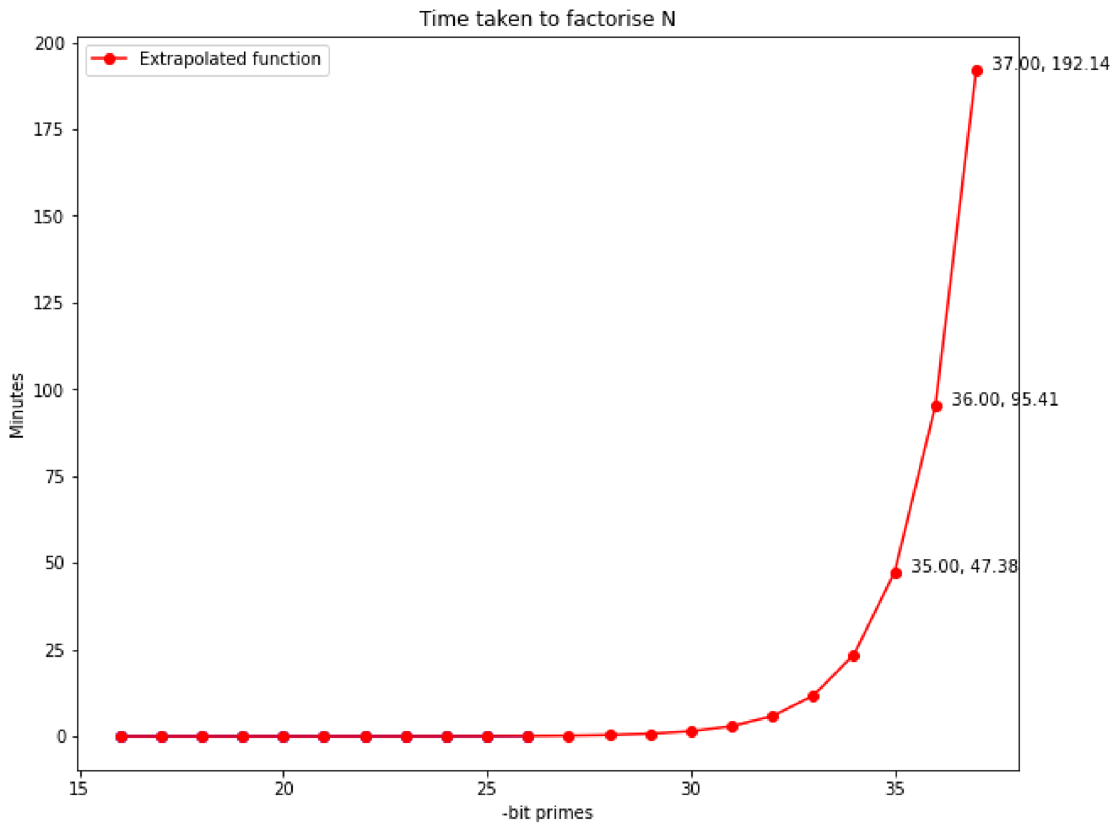
V1 - runs until it is unfeasible

[2]:
```
# Generate primes p, q and input N = p*q to the smallest_function and show the␣
↪average computation time on input
# l = 16 bit primes and N = p*q for k = l, l+1, l+2,...

l = 16 # -bit primes

p = c3.random_prime(l)
q = c3.random_prime(l)
N = p*q

# List to store the number of bit-prime, i.e., k = l, l+1, l+2,... starting␣
↪with l = 16.
bit_primes = []
# List to store the average computation time for smallest_factor(N).
time_taken = []
average_time_taken = 0

# Start time to calculate the time elapsed while running the function␣
↪repeatedly with increasing bit primes.
# So that we can find out how long has it passed to factorise N and modify the␣
↪while loop condition to preventing
# it from runnning too long.
start = time.time()
elapsed = 0
# Just an estimate with 150 seconds to start off with.
# Can modify it to find out the average computation time for greater bit primes.
while elapsed < 150:
    # Use random_prime(l) function to generate a random prime number with␣
↪l-bits.
    p = c3.random_prime(l)
    q = c3.random_prime(l)
```

```
    N = p*q
    # Use %timeit to measure the average computation time of the␣
 ↪smallest_factor(N) function.
    t = %timeit -o c3.smallest_factor(N)
    bit_primes.append(l)
    # Store the time taken as minutes.
    average_time_taken = t.average/60
    time_taken.append(average_time_taken) # t.average to store the average␣
 ↪computation time in time_taken list.
    elapsed = time.time() - start
    l += 1

# Plot two lines in a graph.
plt.figure(figsize = (10,8))

# The first plotted line is the results from the above smallest_factor function␣
 ↪performance test.
x_values = bit_primes
y_values = time_taken
plt.plot(x_values, y_values,'b-o')

# The second plotted line is an estimated function with extrapolation to find␣
 ↪out when the smallest_factor function becomes
# unfeasible. We extrapolated it until 35-bit primes, as it is unfeasible to␣
 ↪factorise N using the smallest_factor function
# starting at 35-bit primes.
x_values_ext = [i for i in range(16, 38)]
y_values_ext = [0.0000000010849*np.exp(0.7*x) for x in x_values_ext]
plt.plot(x_values_ext, y_values_ext, 'r-o', label = 'Extrapolated function')
plt.text(x_values_ext[-3], y_values_ext[-3], "   {:.2f}, {:.2f}".
 ↪format(x_values_ext[-3], y_values_ext[-3]))
plt.text(x_values_ext[-2], y_values_ext[-2], "   {:.2f}, {:.2f}".
 ↪format(x_values_ext[-2], y_values_ext[-2]))
plt.text(x_values_ext[-1], y_values_ext[-1], "   {:.2f}, {:.2f}".
 ↪format(x_values_ext[-1], y_values_ext[-1]))

plt.xlabel("-bit primes")
plt.ylabel("Minutes")
plt.title("Time taken to factorise N")
plt.legend()
plt.show()

print("bits_list: {}".format(bit_primes))
print("time_list: {}".format(time_taken))
print("time taken: {} minutes".format(elapsed/60))
```

```
5.61 ms ± 62.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
9.86 ms ± 152 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
19.7 ms ± 340 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
35.2 ms ± 615 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
92.6 ms ± 1.5 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
255 ms ± 3.61 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
437 ms ± 7.41 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
936 ms ± 13.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.44 s ± 11 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
3.92 s ± 33.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
8.46 s ± 201 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```




```
bits_list: [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
time_list: [9.356273932471161e-05, 0.00016429866052099638,
0.00032841846373464386, 0.0005864626581647566, 0.0015427138394720498,
0.004252027715778067, 0.007284465968786251, 0.015607080360253652,
0.023998964582348157, 0.06539419504946896, 0.14104588768400605]
time taken: 2.7105963786443072 minutes
```

From both the plotted line, we can see that the amount of time taken to factorise N grows exponentially as the bit primes increases. When p and q are 16-bit primes, the average computation time taken to factorise N is approximately 5 ms, when p and q are 24-bit primes, it takes around 1-2 seconds, at the end of the while loop, with p and q are 26-bit primes, it takes around 8-9 seconds to factorise N, hence, it is very hard to factorise N if only N is known and when p, q are large bit primes, i.e., 512-bit primes.

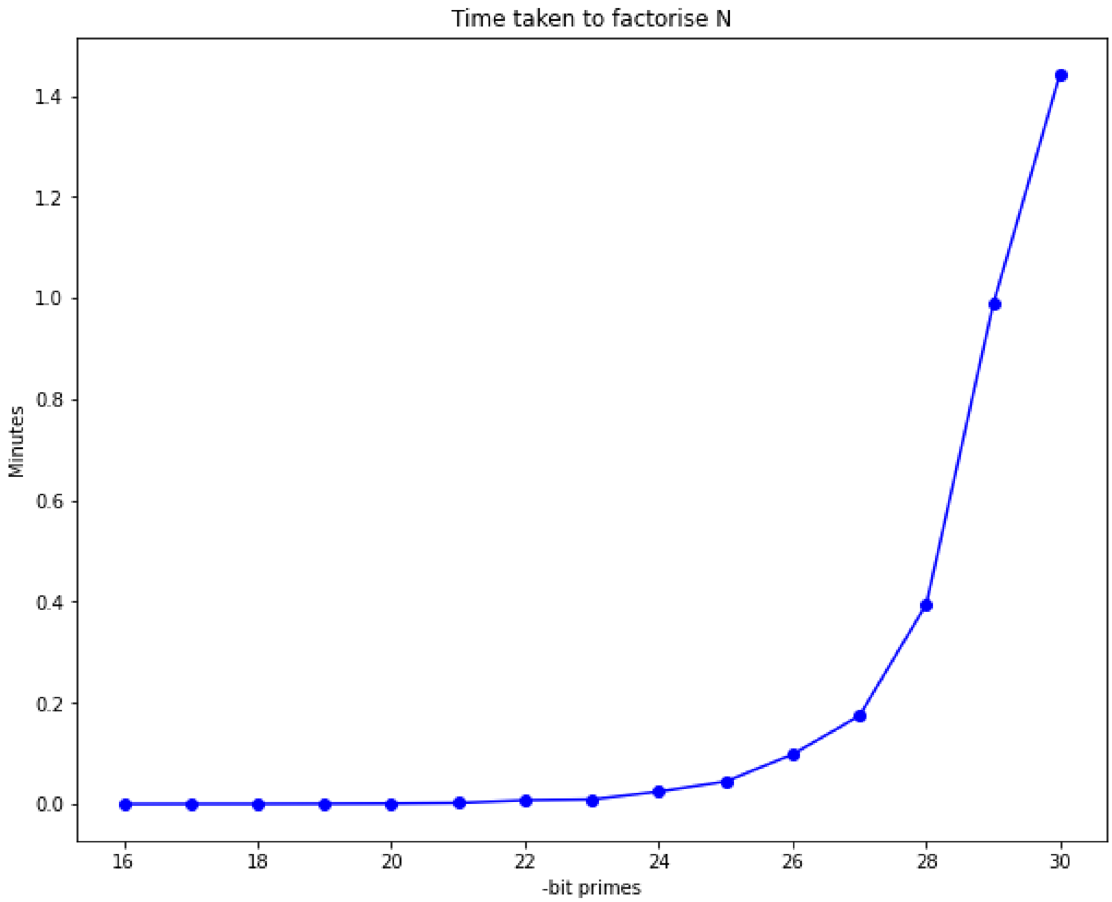
From the extrapolated line, we can say that it is starting to be unfeasible to factorise N using smallest_factor function when p and q are 35-bit primes, as it takes 47 minutes to factorise N and with 37-bit primes, it takes around 192 minutes to factorise N.

V2 - runs for 15 mins

[4]:
```
# Generate primes p, q and input N = p*q to the smallest_function and show the
 ↪average computation time on input
# l = 16 bit primes and N = p*q for k = l, l+1, l+2,...

l = 16 # -bit primes

p = c3.random_prime(l)
q = c3.random_prime(l)
N = p*q

# List to store the number of bit-prime, i.e., k = l, l+1, l+2,... starting
 ↪with l = 16.
bit_primes = []
# List to store the average computation time for smallest_factor(N).
time_taken = []
average_time_taken = 0

# Start time to calculate the time elapsed while running the function
 ↪repeatedly with increasing bit primes.
# So that we can find out how long has it passed to factorise N and modify the
 ↪while loop condition to preventing
# it from runnning too long.
start = time.time()
elapsed = 0
# Let it runs for around 20 minutes.
while elapsed < 15*60:
    # Use random_prime(l) function to generate a random prime number with
 ↪l-bits.
    p = c3.random_prime(l)
    q = c3.random_prime(l)
    N = p*q
    # Use %timeit to measure the average computation time of the
 ↪smallest_factor(N) function.
    t = %timeit -o c3.smallest_factor(N)
    bit_primes.append(l)
    # Store the time taken as minutes.
    average_time_taken = t.average/60
    time_taken.append(average_time_taken) # t.average to store the average
 ↪computation time in time_taken list.
```

```
    elapsed = time.time() - start
    l += 1

# Plot the graph.
plt.figure(figsize = (10,8))

# The plotted line is the results from the above smallest_factor function␣
 ↪performance test.
x_values = bit_primes
y_values = time_taken
plt.plot(x_values, y_values,'b-o')

plt.xlabel("-bit primes")
plt.ylabel("Minutes")
plt.title("Time taken to factorise N")
plt.show()

print("bits_list: {}".format(bit_primes))
print("time_list: {}".format(time_taken))
print("time taken: {} minutes".format(elapsed/60))
```

```
5.22 ms ± 157 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
14.2 ms ± 84.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
18.1 ms ± 237 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
34.7 ms ± 2.02 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
64.5 ms ± 432 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
134 ms ± 1.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
451 ms ± 7.45 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
547 ms ± 12.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.47 s ± 38.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.68 s ± 23.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
5.88 s ± 46.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
10.5 s ± 85.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
23.7 s ± 113 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
59.3 s ± 216 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1min 26s ± 400 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```




```
bits_list: [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
time_list: [8.705232509722314e-05, 0.0002362227992021612,
0.00030241236574060865, 0.0005786338271129699, 0.0010743808285111473,
0.0022378336252378566, 0.007518527878537064, 0.00910945703231153,
0.024435643594534626, 0.044651387188406215, 0.09800695342322191,
0.1745351156663327, 0.39523696110007306, 0.9879941397257859, 1.4432472601178146]
time taken: 26.282700804869332 minutes
```

The above algorithm runs around 20-30 minutes, and it only factorises N until 30 bit primes.