

# The Goldbach Conjecture

## Sections

### Summing Primes

*Here we verify the conjecture for small numbers.*

### The Sieve of Eratosthenes

*A fairly fast way to determine if small numbers are prime, given storage.*



§2. Computer verification has been made up to around  $10^{18}$ , but by rather better methods than the one we use here. We will only go up to:

```
define RANGE 100
```

```
#include <stdio.h>
```

```
int main(int argc, char *argv[]) {
```

```
    for (int i=4; i<RANGE; i=i+2)
```

```
        <Solve Goldbach's conjecture for i 2.1> ;
```

```
}
```

*stepping in twos to stay even*

§2.1. This ought to print:

```
$ goldbach/Tangled/goldbach
```

```
4 = 2+2
```

```
6 = 3+3
```

```
8 = 3+5
```

```
10 = 3+7 = 5+5
```

```
12 = 5+7
```

```
14 = 3+11 = 7+7
```

```
...
```

We'll print each different pair of primes adding up to  $i$ . We only check in the range  $2 \leq j \leq i/2$  to avoid counting pairs twice over (thus  $8 = 3 + 5 = 5 + 3$ , but that's hardly two different ways).

<Solve Goldbach's conjecture for i 2.1>  $\equiv$

```
printf("%d", i);
```

```
for (int j=2; j<=i/2; j++)
```

```
    if ((isprime(j)) && (isprime(i-j)))
```

```
        printf(" = %d+%d", j, i-j);
```

```
printf("\n");
```

This code is used in §2.

# The Sieve of Eratosthenes

*A fairly fast way to determine if small numbers are prime, given storage.*

---

S1 Storage; S2 Primality

---

§1. This technique, still essentially the best sieve for finding prime numbers, is attributed to Eratosthenes of Cyrene and dates from the 200s BC. Since composite numbers are exactly those numbers which are multiples of something, the idea is to remove everything which is a multiple: whatever is left, must be prime.

This is very fast (and can be done more quickly than the implementation below), but (a) uses storage to hold the sieve, and (b) has to start right back at 2 - so it can't efficiently test just, say, the eight-digit numbers for primality.

```
int still_in_sieve[RANGE + 1];
int sieve_performed = FALSE;
```

§2. We provide this as a function which determines whether a number is prime:

```
define TRUE 1
define FALSE 0

int isprime(int n) {
    if (n <= 1) return FALSE;
    if (n > RANGE) { printf("Out of range!\n"); return FALSE; }
    if (!sieve_performed) <Perform the sieve 2.1> ;
    return still_in_sieve[n];
}
```

§2.1. We save a little time by noting that if a number up to RANGE is composite then one of its factors must be smaller than the square root of RANGE. Thus, in a sieve of size 10000, one only needs to remove multiples of 2 up to 100, for example.

```
<Perform the sieve 2.1> ≡
    <Start with all numbers from 2 upwards in the sieve 2.1.1> ;
    for (int n=2; n*n <= RANGE; n++)
        if (still_in_sieve[n])
            <Shake out multiples of n 2.1.2> ;
    sieve_performed = TRUE;
```

This code is used in §2.

§2.1.1.

```
<Start with all numbers from 2 upwards in the sieve 2.1.1> ≡
    still_in_sieve[1] = FALSE;
    for (int n=2; n <= RANGE; n++) still_in_sieve[n] = TRUE;
```

This code is used in §2.1.

§2.1.2.

```
<Shake out multiples of n 2.1.2> ≡
    for (int m= n+n; m <= RANGE; m += n) still_in_sieve[m] = FALSE;
```

This code is used in §2.1.