Documentation for Findprimes.C

All prime number calculations are done by the function primeseek, everything else is just for testing purposes. The function return is a pointer to a dynamically allocated array of primes (type is unsigned 32 bit int right now), the function receives a parameter that tells it what is the largest prime we seek and it returns via reference the number of primes found.

The calculations use sieve of Eratosthenes, which is the oldest since I wasn't interested in the mathematics of prime finding but finding faster coding solutions. Below I present the oprimizations used to speed up the program from several minutes to 2.7 seconds over finding all the primes up to the unsigned 32 bit limit.

At first the program obviously ignores even numbers since they are all not primes after 2 itself. Every number during internal calculations are odd, so the sieve only contains odd numbers.

Secondly, besides the sieve, I designed a “stamp”. What is it? We know that 3 and 5 are primes, so we can design a 3*5 long stamp:

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29

The greyed out numbers are all divisible by 3 or 5. This creates the stamp:

	▨	▨		▨			▨			▨		▨	▨	

This stamp can be placed onto any 15 odd numbers starting from 30*n+1

31	33	35	37	39	41	43	45	47	49	51	53	55	57	59

3001	3003	3005	3007	3009	3011	3013	3015	3017	3019	3021	3023	3025	3027	3029

The greyed out numbers are surely not primes. The white background ones can be primes or can be divisable by other numbers, for example 49 is 7*7.

The program uses a 3 * 5 * 7 * 11 * 13 big stamp. Out of its 15015 numbers, only 5761 are possible primes, the others are jumped over.

Third optimization was choosing the proper size of the sieve. Since every time a new sieve is used, a square root must be calculated, and then all primes bigger than 13 and smaller than square root of the biggest number in the sieve must be divided once. If we double the sieve size, we need half as many square roots and divisions to find the same amount of primes, so common sense suggests to increase sieve size as memory allows. Which turned to be wrong. Division is faster than memory access, so the sieve must be as small as the L1 cache of the computer. As my computer has 2 way 32K L1 cache, I use 32Kb memory sieves. Using even 33K long sieves slows the program significantly.

The fourth optimization can be seen between lines 83 and 90: bitpacking. Instead of using a byte for one number in the sieve, I use a bit, so the 32Kb sieve can actually hold 8*32K numbers. This way I decrease the number of divisions and square roots by 8 without getting cache misses.

The fifth optimization is seen as a macro at line 94-97 which is called twice, for seemingly no reason. This is the sieve reader that finds which numbers turned out to be primes and place them on the prime number list. It loops over the sieve with two exit conditions: end of sieve and end of stamp. In case of end of stamp, we must jump to the beginning of the stamp again. Calculating 2 exit conditions is expensive, so I first – outside of the loop – calculate which condition will arrive first, and loop only for that condition.

The sixth optimization can be seen inside the macroed sieve reader at line 94-97. What it does is:

- mindlessly add the next non-stamped number to the top of the prime list, and increase the list top
- check the sieve and if the number wasn't a prime, decrease the list top, effectively deleting the number the previous line just added

This is again against common sense, which would suggest "check the sieve and if the number is a prime, add it to the list, increase list top". The common sense answer is much slower, because it contains brancing. The condition is evaluated and if it's false, the execution must jump over the "add to the list, increase top" part. Execution jumps are very slow, since they mess up the instruction pipeline. The optimized version always performs the "add to the list, increase top" part, so it's always pipelined, The conditional decrease of a single number can be done without execution jump, using the MOVC instruction and the C compiler is smart enough to realize that and compiles my optimized code accordingly.