How to search for 26 primes in arithmetic progression?

by **Jarosław Wróblewski**

Version 3, May 23, 2008
http://www.math.uni.wroc.pl/∼jwr/AP26
jwr@math.uni.wroc.pl

0. Abstract

In this paper I am presenting the main tricks I have applied to write the computer program which was used to find the first known arithmetic progressions of primes of length 24 (January 18, 2007) and 25 (May 17, 2008). The same idea can be used to search for an arithmetic progression of 26 primes. I am giving a general idea of searching for long arithmetic progressions of primes, as well as a more detailed description of a specific search refering to a working program searching for AP26 I have written.

1. Copyleft notice

Feel free to distribute printed or soft copy of this document as long as you do not conduct any misuse of it. If you understand what I mean by a misuse, no explanation is needed. If you don't understand what I mean by a misuse, no explanation would help.

In particular, a **copy** can be **left** on your website if it is going to serve a good purpose.

2. Introduction

Although I can write a computer program, I am not a profesional programmer, I feel more like an algorithm designer.

I was able to design a fairly efficient algorithm and to write some working programs, but in order go get a good use of the algorithm, skillful programmers must take over the details of the algorithm and work on them.

At this point I cannot do anything more than just share my ideas with people interested in searching for arithmetic progressions of primes.

Since `PrimeGrid` administrators are considering to include AP26 search, please do not do anything that would interfere with those plans, in particular please do not start a wide search without contacting them. They may need, however, some help of a skillfull programmer, who would understand the details of the AP26.h code and would like to deticate time to help optimize it and do whatever preparations for running it are needed. Please contact them if you would like to help. Since I have no imagination what programming issues come to play when launching a distributed project, I cannot tell you what is really needed - they know that better.

3. The two tricks

In principle it is obvious how to search for prime arithmetic progressions:

Step 1. Perform some kind of sieving to rule out progressions with a term divisible by a small prime.

Step 2. Check primality of terms of any sequence which passes Step 1.

While there is nothing fancy about Step 2, Step 1 is the key of any search.

We assume that the progression difference is fixed in a program segment.

The first trick

It seems that there are 2 rules which should apply when setting up a sieve:

Rule 1. Use as much RAM as you have access to.

Rule 2. Use the smallest primes for sieving.

The first trick is to violate both of the above rules.

It is well known that using hard drive for storing temporary data is not a good idea, since RAM is much faster and frequent usage of hard drive virtually stops the program.

Similarly, accessing data stored in RAM is very slow compared to using processor's cache.

Hence we do not want to use significant amount of memory. Therefore the first stage of sieving should work in a deterministic way, using as simple operations as possible.

Those remarks get ride of Rule 1.

Regarding Rule 2, we are searching for sequences with differences divisible by 2,3,5,7,11,13,17,19,23.

Suppose we want to do a sieving for sequences of length 24. Sieving is just checking the residue of the first term of the progression.

Sieving mod 23 means we have one bad residue and 22 good ones - hence by using 23 for sieving we get rid of 4.35% candidates for progressions.

Sieving mod 53, with progression difference non-divisible by 53, means we have 24 bad residues and 29 good ones, hence using 53 reduces number of candidates by 45.3%.

Given that there is a very limited number of primes we can use for sieving at this stage, we prefer to use 53 rather than 23. In fact it seems profitable to use 2,3,5 and not to use 11,13,17,19,23, while using or not using 7 is neutral - it can be used to manipulate the search range. It was used in AP24 and AP25 search, but I think it shouldn't be used in AP26 hunt, making room for 59 to be used instead.

AP26 search - Stage 1

To set up an AP26 search we assume the following:

$1°$ Progression difference D is divisible by $23\#$ and is **not** divisible by any of the following primes: 29, 31, 37, 41, 43, 47, 53, 59.
D is fixed during the whole search segment.

$2°$ We use the following primes for sieving at this stage: 2, 3, 5, 29, 31, 37, 41, 43, 47, 53, 59.

$3°$ We do sieving for sequences of length 24 in a specific way described below.

Let p be one of the primes listed at $2°$. Residue r modulo p will be called good if no of the numbers r, $r+D$, $r+2D$, ..., $r+23D$ is divisible by p, otherwise it will be called bad.

For $p=2,3,5$ all non-zero residues are good.

For the remaining p's all the good residues are D, $2D$, $3D$, ..., $(p-24)D$.

We make $p=29$ an exception, where we consider $3D$ to be the only good residue.

Let $MOD = 2 \cdot 3 \cdot 5 \cdot 29 \cdot 31 \cdot 37 \cdot 41 \cdot 43 \cdot 47 \cdot 53 \cdot 59 = 258559632607830$.

Then a residue R modulo MOD is good iff it reduces to a good residue modulo any prime divisor of MOD.

The objective of this stage is to generate all good residues modulo MOD.

Let n_0 will be the residue with the following properties:

$$n_0 \equiv 1 \,(\mathrm{mod}\ 2)$$

$$n_0 \equiv 1 \,(\mathrm{mod}\ 3)$$

$$n_0 \equiv 1 \,(\mathrm{mod}\ 5)$$

$$n_0 \equiv 3D \,(\mathrm{mod}\ 29)$$

$$n_0 \equiv D \,(\mathrm{mod}\ p)$$

for $p=31,37,41,43,47,53,59$.

Let moreover

$$s_3 \equiv 1 \,(\mathrm{mod}\ 3)$$

$$s_5 \equiv 1 \,(\mathrm{mod}\ 5)$$

$$s_p \equiv D \,(\mathrm{mod}\ p)$$

for $p=31,37,41,43,47,53,59$. and let all the above s's be divisible by all prime factors of MOD except for congruences explicitly listed above.

Then all the good residues modulo MOD can be obtained by the formula:

$$R = n_0 + i_3 s_3 + i_5 s_5 + i_{31} s_{31} + \ldots + i_{59} s_{59}$$

with

$$i_3 = 0,1$$

$$i_5 = 0,1,2,3$$

$$i_{31} = 0,1,2,3,4,5,6$$

$$i_{37} = 0,1,2,3,4,\ldots,12$$

$$\ldots\ldots\ldots\ldots$$

$$i_{59} = 0,1,2,3,4,\ldots,34$$

Those residues can be generated in nested loops using addition, comparison and subtraction only.

Conclusion: Complete set of good residues modulo MOD can be delivered to the next stage of the program at a very high speed.

AP26 search - Stage 2

Now we are given a good residue R modulo MOD. As a consequence the sequence

$$a_i = R + iD, \quad i = 0,1,2,\ldots,23$$

is sure to have no terms divisible by primes listed in 2°. Does it have a term divisible by 61? This is purely determined by $R \pmod{61}$, so we could compute $R \pmod{61}$ and look in a precomputed table if it is good or bad residue modulo 61. If it is bad, we can abandon the sequence. If it is good we can do the same test for 67 instead of 61.

The problem is that we have to perform division, which is rather costly operation. OK, we have to make a division, but once we do it, can we make more use of it? And here comes...

The second trick

A good residue R will not represent just the single sequence

$$a_i = R + iD, \quad i = 0,1,2,\ldots,23$$

but it will represent 64 sequences

$$a_i = b \cdot MOD + R + iD, \quad i = 0,1,2,\ldots,23$$

with b running from 0 to 63. Hence computing $R \pmod{61}$ determines which of the 64 sequences are good (contain no term divisible by 61) and which are bad. This piece of information, depending purely on $R \pmod{61}$, can be precomputed and stored as a 64-bit word - each bit telling the whole story about the corresponding sequence.

So we compute $R \pmod{61}$, take a 64-bit word from a precomputed table, and we have 64 bits telling which of the 64 sequences are worth consideration.

Then we compute $R \pmod{67}$, take another 64-bit word from precomputed table and bitwise AND it with the previos one. Then the bit 1 will appear only on positions corresponding to sequences which passed both tests.

The result of the bitwise AND is then bitwise AND-ed with the corresponding word for 71 and so on.

Note: b can run through any other interval of 64 consecutive integers to continue the search, e.g. 64-127, 128-191, ...

4. General searching program

While planning a search one has to decide the following:
Q1. Which small primes must be divisors of the progression difference?
Q2. What range should the first term of the progression come from?
Q3. How long a sequence do you expect to find?
After deciding on **Q1** and assuming the progression difference is divisible by $p_1 p_2 \ldots p_x$, you have to pick a set of primes which will determine the range of the first term.

Those primes $q_1, q_2, \ldots, q_y$ should be selected as follows:
Include 2,3,5.
Include or not 7.
Do not include any other p_i's.
Include the smallest primes not being p_i's.
The range of the first term will be $64 \cdot MOD$, where $MOD = q_1 q_2 \ldots q_y$.
You can adjust that range by deciding whether to include 7 and how many other primes to take.
Remember that progression difference cannot be divided by any of the q_j's, which are not p_i's.
Example:
While planning AP24 search I took 2,3,5,7,11,13,17,19,23 as p_i's and 2,3,5,7,29,31,37,41,43,47,53 as q_j's.
That gave the range for the first term

$$64 \cdot MOD = 64 \cdot 2 \cdot 3 \cdot 5 \cdot 7 \cdot 29 \cdot 31 \cdot 37 \cdot 41 \cdot 43 \cdot 47 \cdot 53 = 1.9633 \cdot 10^{15} \, .$$

The closest other options were:
Skip 7:

$$64 \cdot MOD = 64 \cdot 2 \cdot 3 \cdot 5 \cdot 29 \cdot 31 \cdot 37 \cdot 41 \cdot 43 \cdot 47 \cdot 53 = 2.80471 \cdot 10^{14} \, ,$$

which seems too low for the search.
Take 59 instead of 7:

$$64 \cdot MOD = 64 \cdot 2 \cdot 3 \cdot 5 \cdot 29 \cdot 31 \cdot 37 \cdot 41 \cdot 43 \cdot 47 \cdot 53 \cdot 59 = 1.65478 \cdot 10^{16} \, ,$$

which seems too large for AP24 or AP25 search, but is appropriate for AP26 search.

Regarding **Q3**, one must decide for how long progression to do the sieving. However, this doesn't have to match exactly the length of searched sequence. Sometimes you may want to do sieving for shorter sequence and count that the sequence extends behind the sieved segment. Or you can sieve for a longer sequence, and then a shorter one has more room to fit there. But to use sieving with length different than expected sequence, you should know why you are doing so.

So let's say that you have decided to sieve for sequences of length L and that the difference of the progression is going to be D.

If there is no reason to do any modifications to the searching procedure, stage 1 of the program delivers all the numbers n with the following properties:
1° $0 < n < MOD$
2° No of the numbers n, $n+D$, $n+2D$, $n+3D$, $\ldots$, $n+(L-1)D$ is divisible by any of the q_j's.
Property 2° holds not only for n, but also for all the numbers of the for $n + b \cdot MOD$.

Let $r_1, r_2, r_3, \ldots$ be all the primes which are not p_i's or q_j's. Stage 2 of the search uses precomputed tables to select sequences with terms non-divisible by $r_1, r_2, r_3, \ldots$ in a manner described in the previous section.

5. AP26 search

Now let's refer to my sample implementation, which can be used for AP26 search. All the files are in

`http://www.math.uni.wroc.pl/~jwr/AP26/AP26.zip`

In those files I have provided a working routine, which does the computations, constructed in such a way, that it can be used as a Black Box, without looking inside or changing anything. Optimizations are welcome, but not neccessary, the routine can be used as is.

The routine is called

`void SearchAP26(int K, int SHIFT)`

and is placed in `AP26.h` with includes in

`CONST.H    IF.H    MAKES.H    SITO.H`

To operate, the routine needs two other routines:

`int PrimeQ(long long value)`

which tests `value` for primality and which is in two versions in

`PrimeQ32.h    PrimeQ64.h`

and the routine

`void ReportSolution(int AP_Length,int difference,long long First_Term)`

which you must write yourself to suit your needs.

ReportSolution is called by SearchAP26 whenever AP10 or longer is encountered, the progression found has the difference `difference*23#`, the other two parameters have their exact meaning.

In this routine, you have to define the action to be taken, depending on `AP_Length`. For testing purposes it may be reasonable to store all the sequences of length at least 10-14, for serious run keeping track of sequences shorter than 20 seems rather pointless. Perhaps it would be reasonable to keep track of AP22 and longer, but it depends on how you handle the results. Certainly storing AP23's is a must as they will frequently break AP23 records.

How to schedule execution of SearchAP26? Runs of this routine are completely independent. You need to call

`SearchAP26(K,shift);`

with all integers K and with all shift divisible by 64, starting from 0.

For $K > 318,000,000$ there will be overfow problems very easy to fix by editing lines 40-49 of AP26.h, but I think so large K will never be reached.

From technical point of view, the order of execution doesn't matter, but the common sense tells: start searching where the solution is more likely.

That doesn't have to be strictly observed, but my sugestion is to start shift=0, K from 1 going up.

When K reaches 10,000,000, start shift=64, K from 1 going up, but continue shift=0 at the same time. So the computations at this stage should include both shift=0 or shift=64, with shift=64 being 10,000,000 behind (if it is 8,000,000 or 11,000,000 at some moments that's not a big deal).

Now, each next shift should be started with a delay of 7,000,000.

That is, when you reach
shift=0, K=17,000,000
shift=64, K=7,000,000
it is time to start shift=128 from K=1.

After reaching
shift=0, K=24,000,000
shift=64, K=14,000,000
shift=128, K=7,000,000
you start shift=192 and so on.

As I have tested the current version of the program, it seems that the difference between 64 and 32-bit computers is not so severe as before, but is still large.

If K is divisible by one of the primes in the range 29-59, this K is ignored, i.e. SearchAP26 quits instantly. Those are just over 18% of all K's.

Ordinary K's were taking, depending on processor, 7-12 minutes on 64-bit computers I have tested and some 55 minutes on a 32-bit machine. If K is divisible by 61, this time is increased by some 20% and if a few next primes (67, 71, ...) divide K it can be even longer.

Although one program can safely call SearchAP26 with various shift, this seems to be pointless from the point of view of assigning the ranges.

To illustrate how to use Search AP26, look at the three *.c files. While I used timing* for timing and testing purposes (timing32.c should rediscover AP25 and timing64.c one of the AP24's), a sample mechanism managing search ranges is in AP26-64.c

AP26-64.c uses search range in the format
KMIN KMAX shift
For example, if I was going to distribute BY HAND the begining of the search with ranges of length 100 I would run
AP26-64 1 100 0
AP26-64 101 200 0
AP26-64 201 300 0
and so on... AP26-64 reads the range from commandline arguments, if they are not there, it attempts to read them from a file, which is being updated after each K computed. So AP26-64 with no arguments would restart a search.

This is enough to run program by hand or with a shell script on a local network.

All this stuff does the job for explaining how to conduct a search with a ready routine, but so far doesn't address the details of the algorithm and its implementation.

We assume that the progression difference is equal to
`STEP=K*23#` (line 37 of AP26.h)
where K is not divisible by 29,31,37,41,43,47,53,59 (lines 25-34 of AP26.h and lines 2-9 of CONST.H).

STEP was denoted by D in Section 3.

Although the purpose is to find AP26, I have decided to sieve for length $L=24$. However the prime 29 is treated in an exceptional way. In fact the routine searches for such AP24 of the form $(a_0, a_1, a_2, \ldots, a_{23})$ that a_{-3} and a_{26} ARE divisible by 29. This will

capture all AP26 on the way of the search.

S3, S5, S31, S37, ..., S59 have their meaning as s_p described in Section 3 and n0 stands for n_0. While S3 and S5 do not depend on K, numbers S31, ..., S59 are computed from PRES2, ..., PRES8.

On line 69 of AP26.h there is MAKES.H included. It precomputes tables used for sieving in Stage 2.

Let us take OK61 and OKOK61 as an example. Ignore SHIFT for the moment by thinking it is 0.

OK61[R] must tell, whether R is good or bad residue mod 61.

R is a good residue iff the sequence R, R+STEP, R+2*STEP, ..., R+23*STEP contains no term divisible by 61 and it is marked as OK61[R]=1.

Otherwise R is bad and marked by OK61[R]=0.

Note that you cannot list good residues not knowing whether STEP is divisible by 61 or not, but you can list bad residues, which are 0, 60*STEP, 59*STEP, ..., 38*STEP (lines 1-2 of MAKES.H). This list contains 24 bad residues if STEP is not divisible by 61, or residue 0 repeated 24 times otherwise. In any case it is what we need.

Go to lines 167-170 of MAKES.H, where OKOK61[R] is computed.

OKOK61[R] is a 64-bit word, whose b-th bit tells whether R+b*MOD is a good residue (bit=1) or bad (bit=0).

Now we are on line 70 of AP26.h and we have to go through all i_3 - i_{59} as described in Section 3. We take i_3 - i_{41} as outer loops and i_{43} - i_{59} as inner loops. Outer loops commands are executed not very frequently, so there is no harm in putting in line 84:
`n43=(n0+i3*S3+i5*S5+i31*S31+i37*S37+i41*S41)%MOD;`
with multiplications and divisions. Such construction allows to make frequent checkpoints.

Loops defined in lines 72-78 go through all the specified systems (i3,i5,i31,i37,i41). Those systems define independent loops and can be executed in any order, shared between computers, crashed and restarted. This perhaps doesn't constitute crucial issue here, but some other AP searches can have much longer segments, and this becomes an important issue.

There are 4 inner loops that involve addition, subtraction and comparison only (ignore commands with r61, s61, m61 through 97 for the moment).

At line 110 there are numbers n59 being delivered at high speed (have a look at lines 160 and 169-170, where they are being computed in the most inner loop).

They have the property that the sequence
n59, n59+STEP, ..., n59+23*STEP
has no term divisible by any of the q_j's and is correctly placed with respect to numbers divisible by 29. If a multiple of MOD is added to all the terms of the above sequence, this property still holds.

At line 111 we have SITO.H included. It does all the sieving with the 64-bit words described in Section 3.

Note that line 1 of SITO.H should contain
`sito=OKOK61[n59%61];`

but it contains
`sito=OKOK61[r61];`
instead. That is because I wanted to avoid costly division and kept track of r61=n59%61 in the meantime, using additive/comparitive operations. I used this trick up to 97. I am not sure whether it is optimal (it may depend on a computer), but optimum is somewhere there. This is a place for an optimizing tune up. SITO.H does 64-bit sieving with primes up to 331. This is another room for a tune up - going too high may cost memory, which we do not want to go into RAM. Also most n59's do not make it through the first 10 primes and quit after all 64 bits are zeroed.

At line 120 n is computed. n is the candidate for the first term of the sequence, but since the sequence is being extended back and forth, it may end up being non-first term or a non-term.

Some more primes are sieved in IF.H included on line 122 and in divisibility checks on lines 123-128 (perhaps the checks should go ahead of IF.H, but I do not think it matters much).

Lines 129-154 are verifying the candidate sequence
n, n+STEP, n+2*STEP, ..., n+23*STEP
using a primality test.

And that completes the desciption of the algorithm implemented in AP26.h.

6. What is missing?

Regarding AP26 search, the assumption is that no primes in the range 29-59 are divisors of the progression difference. Those missing differences can be searched as well, but they require more programming. Here is an outline of the strategy I would use if I was forced to conduct such a search. Perhaps a better way can be found with better programming skills than mine.

The basic AP26 search assumes p_i's to be 2,3,5,7,11,13,17,19,23 and $q'_j s$ to be 2, 3, 5, 29, 31, 37, 41, 43, 47, 53, 59 with sieving of length L=24 and special treatment of the prime 29.

To capture progressions with differences divisible by 29, I would write a program assuming:
p_i's : 2,3,5,7,11,13,17,19,23,29
q_j's : 2, 3, 5, 31, 37, 41, 43, 47, 53, 59, 61
sieving length L=26.

To capture progressions with differences divisible by 31, I would write a program assuming:
p_i's : 2,3,5,7,11,13,17,19,23,31
q_j's : 2, 3, 5, 29, 37, 41, 43, 47, 53, 59, 61
sieving length L=24 with special role of prime 29.

To capture progressions with differences divisible by 37, I would write a program assuming:
p_i's : 2,3,5,7,11,13,17,19,23,37

q_j's : 2, 3, 5, 29, 31, 41, 43, 47, 53, 59, 61
sieving length L=24 with special role of prime 29.

And so on, up to a program running through sequences with difference divisible by 59. This would make 8 separate programs - a better programmer could include it in one or two programs, I think.

However this would leave outside the search sequeces with difference divisible by two of the primes: 29, 31, 37, 41, 43, 47, 53, 59, 61. Each pair requires separate setup. Note that 67 becomes one of the q_j's.

Then one would have to take care of the case where the progression difference is divisible by three of the primes: 29, 31, 37, 41, 43, 47, 53, 59, 61, 67.

Each of the above programs would call for a different tune up for maximum speed. A balance between program performance and programmers time spend on tuning many programs should be found somewhere.

This is not neccessary if we just want to find any AP26, since we can just ignore differences nondivisible by primes 29-59, but certainly including all possible differences would increase chance of success.

So the first missing thing is extending AP26 search for differences that are currently being skipped.

Another thing, which could serve well to many users, is a program which could handle all (or most) possible searches using the algorithm. Note that after we specify p_i's, q_j's and length L of sieving, there is not much room for programmer's creativity left - perhaps a well written program could conduct a search when a user specifies all the above mentioned parameters.

I have used 64-bit words in Stage 2. I have experimented with 128-bit words, but the results were very poor, so either I did something wrong or that is not the way to do. I have done no experiments with 32-bit words.

7. Solutions found and open problems

Currently there are 10 known arithmetic progressions of primes of length at least 24 (not counting AP24's included in AP25) and all have been found by my program:

```
6171054912832631 + 366384*23#*n, for n=0 to 24
515486946529943 + 136831*23#*n, for n=0 to 23
1368212299836997 + 148682*23#*n, for n=0 to 23
468395662504823 + 205619*23#*n, for n=0 to 23 (first known AP24)
6173646951801709 + 587522*23#*n, for n=0 to 23
401516288036293 + 608239*23#*n, for n=0 to 23
3120744164655323 + 851923*23#*n, for n=0 to 23
1630799846252777 + 985151*23#*n, for n=0 to 23
1564588127269043 + 1249750*23#*n, for n=0 to 23
1606021011322579 + 3490622*23#*n, for n=0 to 23
```

I have found the first known AP24 using computers in students labs in Mathematical Institute of Wrocław University.

All the remaining of the above progressions were found by Raanan Chermoni, who was running my code using computers he had access to.

For the updated list of current records see

`http://hjem.get2net.dk/jka/math/aprecords.htm`

8. Acknowledgements

The following people inspired me to activities, which finally led to creating this document.

Jean-Charles Meyrignac:

by maintaining his website *Computing Minimal Equal Sums Of Like Powers*, turning my attention to *Al Zimmermann's Programming Contests* and being my team partner during a few of those contests (including *Prime Generating Polynomials Contest*), inspired me to deveplop basic programming abilities and algorithm designing skills.

Al Zimmermann:

by creating *Al Zimmermann's Programming Contests* further inspired me to develop my interest in computer aided solving math problems. *Prime Generating Polynomials Contest* attracted my interest to arithmetic progressions of primes.

Jens Kruse Andersen:

by maintaining his website *Primes in Arithmetic Progression Records* gave me motivation to develop my algorithm searching for primes in arithmetic progressions.

Raanan Chermoni:

by taking over the AP24 search with my program, at the moment when I was convinced that a single AP24 is everything my program could find and was about to give up the search. Finding many other AP24's and the first AP25 gave me the final motivation to start working on a written presentation of my algorithm.

9. Examples

In this section I am describing how to set up another search and give a detailed description of another working sample program.

The first step is to go to

`http://hjem.get2net.dk/jka/math/aprecords.htm`

and to analize record tables to find a weak record to improve. The oldest record, which can still be improved is

AP19 with minimal known start:

`8297644387 + 431*19#*n,  0..18,  1984,  Paul Pritchard`

Although the algorithm assumes fixed progression difference and in the current form cannot be used to search for sequences with very small first term, in this case the first term to be improved is still large enough for a reasonable search.

The first thing to decide while planning the search is the list of forced prime divisors of the progression differnce, described before as p_i's.

The best choice seems to be 19#, i.e. p_i's are 2,3,5,7,11,13,17,19.

As for selecting q_j's, the range of the first term is the only factor, which we take into account. We want the search range to be somewhere below $8297644387 \approx 8.3 \cdot 10^9$.

So we have to select the set of primes with the following properties:

2,3,5 are included,

7 is included or not,

some number of **the smallest primes** among 23, 29, 31, ... is included,

the range for the first term, which is 64 times the product of all the selected primes is less than $8297644387 \approx 8.3 \cdot 10^9$, but the largest possible.

By trial and error we decide on 2,3,5,23,29,31,37. This gives MOD=22951470 and 64*MOD=1468894080 $\approx 1.47 \cdot 10^9 < 2^{31}$ and we will be able to use 32-bit integers for some of the variables (in fact MOD$< 2^{30}$ would be enough for that). This however requires much care to avoid an overflow in 32-bit computations.

Also note that 5*(64*MOD) is still below $8.3 \cdot 10^9$, so it makes sense to use SHIFT: 0, 64, 128, 192, 256, 320 to cover first term range up to 384*MOD.

Finally we must decide on the length L of sieved sequence and $L = 19$ seems the only sane choice here.

We must be aware that the search will skip progressions with differences divisible by 23, 29, 31 or 37 (those would require separate programs) as well as progressions with very small first term.

First we create file CONST.H which at the begining contains the main constants used for the search. They are:

- PRIM19 = 19#
- PRIME1 through PRIME4 obvious
- MOD = 2*3*5*23*29*31*37
- N0 - the unique number with the following properties:

N0 is between 0 and MOD

N0 is divisible by 30

N0-PRIM19 is divisible by 23,29,31,37.

- S3 - the unique number with the following properties:

S3 is between 0 and MOD

S3-1 is divisible by 3

S3 is divisible by 5,23,29,31,37.

- S5 - the unique number with the following properties:

S5 is between 0 and MOD

S5-1 is divisible by 5

S5 is divisible by 3,23,29,31,37.

- PRES1 - the unique number with the following properties:

PRES1 is between 0 and MOD

PRES1-PRIM19 is divisible by 23

PRES1 is divisible by 3,5,29,31,37.

- PRES2, PRES3, PRES4 - similarly with 29,31,37 instead of 23.

The sample code can be found in

`http://www.math.uni.wroc.pl/∼jwr/AP26/AP19.zip`

and it is written in a manner similar to the code in AP26.h. Note that there are also some differences in corresponding include files. Compare the two codes and try to understand all the differences.

To test-run the program I have used AP19-64.c which after being compied to AP19-64 can be called as

`AP19-64 KMIN KMAX SHIFT`

I have tested it under 64-bit Linux on

```
model name      : AMD Athlon(tm) 64 Processor 3000+
cpu MHz         : 1809.302
cache size      : 512 KB
```

Computing a range of K's of the length 1000 was taking abot 8 seconds.

I have used 6 computers, one for each SHIFT listed above. Firstly I have scheduled a 2+ hour test with K up to 1,000,000.

Befere the test was finished I have launched range 1,000,001-5,000,000 on other 6 computers.

This happened to be a lucky shot, since I got almost immediately:

`2027663243 + 1002606*19#*n, n=0..18`

At this point I have killed all 12 running programs.

The old AP19 was of course rediscovered and before I killed the programs there were 10 AP17's found, no AP18. The first six-pack of programs was at about 25% of their run, over half an hour each.

As it happened, I have used about 3-4 CPU hours of a single computer before finding the result, but that was due to a lucky schedule. With normal search schedule, where K's are computed in order, it should take about 13-14 CPU hours to arrive at the above result.

Homework for the reader

Exercise 1. Find an AP19 with first term smaller than $2027663243 \approx 2.03 \cdot 10^9$.

Exercise 2. Find an AP20 with first term smaller than $68469367129 \approx 6.85 \cdot 10^{10}$.

Exercise 3. Find an AP21 with first term smaller than $142072321123 \approx 1.42 \cdot 10^{11}$.

Exercise 4. Find an AP22 with first term smaller than $1365385579783 \approx 1.37 \cdot 10^{12}$. Note that this record is an accidental byproduct of another search, where AP with minimal start was not the goal.

Hints: For Exercise 1 you can use the attached program from AP19.zip. You can also write a version assuming progression difference divisible by 23, 29, 31, 37. For the remaining exercises you have to write your own program or make major modifications, otherwise the search is not going to be efficient.

I am refusing to cooperate in any AP search aimed at finding AP of length 19-22 and having smallest known start - this is homework for the readers, I am not going to solve it myself nor help in solving it :-)))

I will be happy to clarify the presentation of the algorithm by filling in more details you find unclear, but I will not get deeper into issues specific for homework exercises.

My goal is to attract more people to AP search, and to see more people learning how to utilize the algorithm I have worked out. I know from my own experiece that finding new records, even minor ones at the beginning, gives the best motivation for further work. That's why I do not care who is going to set those records, while I do care much about giving some fun to others. I would be delighted to see other people making a good use of the ideas I am trying to present in this paper.

Whenever you solve one of the exercises above, check on
`http://hjem.get2net.dk/jka/math/aprecords.htm`
whether it sets a new record. If that is the case, submit the solution to Jens Kruse Andersen, `jens.k.a@get2net.dk`, who is doing a great job maintaining the website and keeping track of all the records.

Have fun and good luck!