
PrimeLister

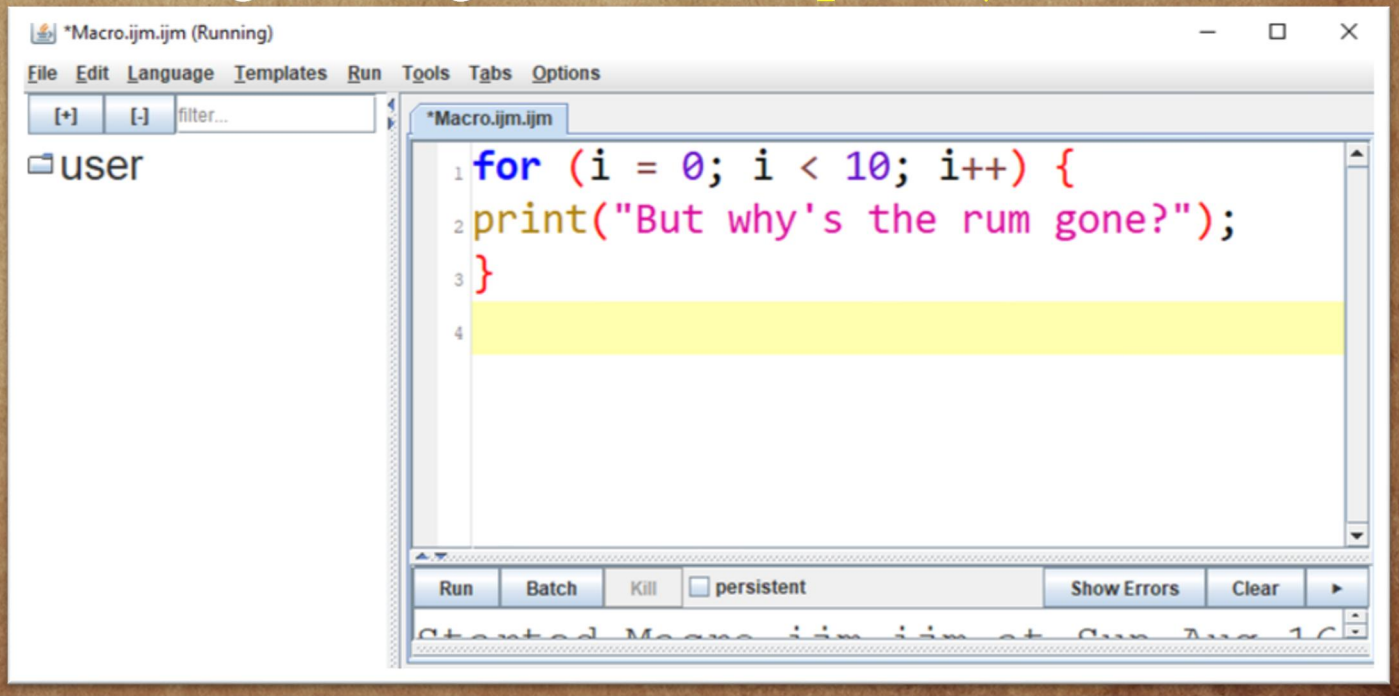
- In this tutorial we will code a gorgeous, elegant and powerful prime number generator in „ImageJ“ - practically a dark elf assassin of the PNGs
- Export that code to C++
- Identify and avoid pitfalls
- Optimize the code up to 19 million primes per second.

We need:

- ImageJ/FIJI (I recommend FIJI)
- A C++ompiler (Dev-C++, Visual Studio, both available for free)
- Microsoft Excel (or something similar)

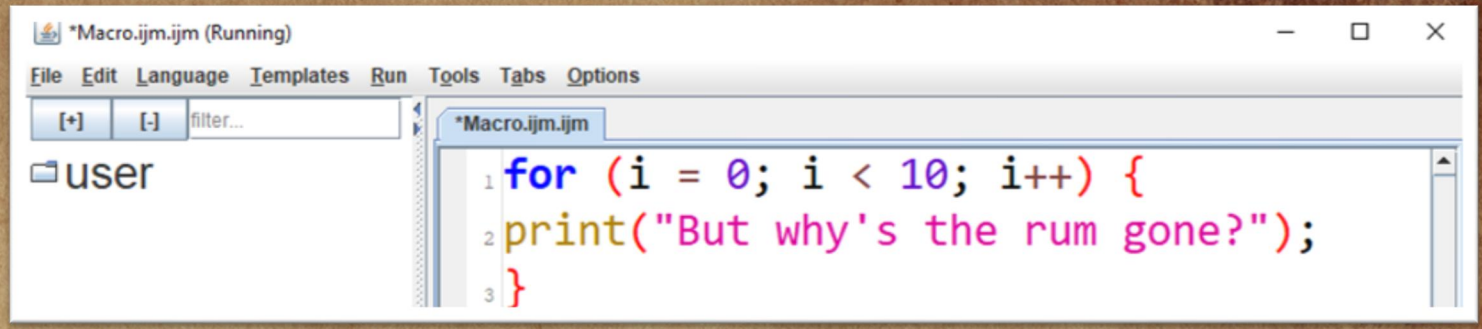
ImageJ / FIJI – why I just love it

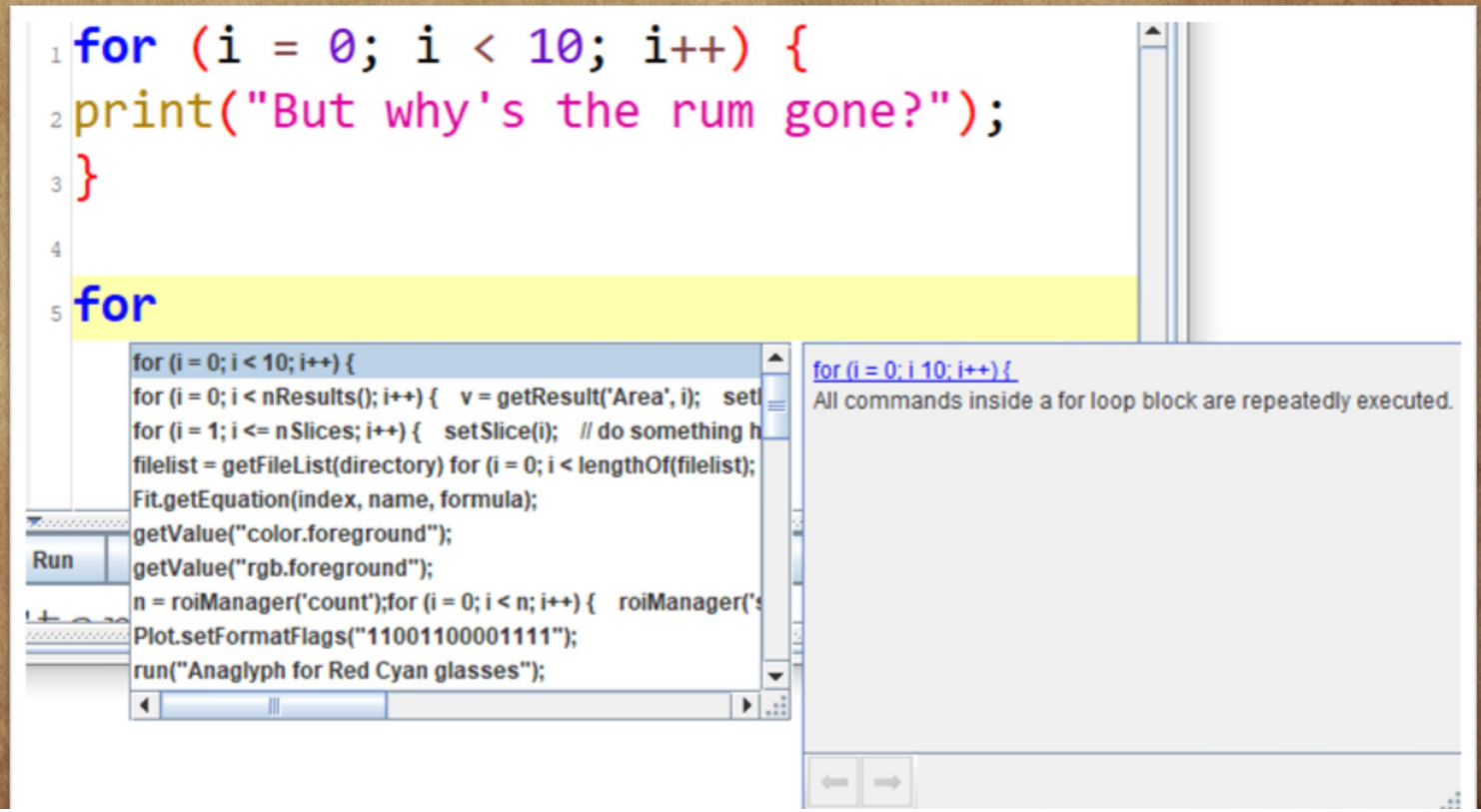
- There are hundreds of programming languages like Java(Script), C/++/#, Python, Visual Basic, Delphi, Pascal, Go, Julia etc., each one better than the other, each one more or less perfect to start with.
- Almost no review mentions ImageJ / FIJI (recursive acronym for "Fiji Is Just ImageJ").
- ImageJ / FIJI (from here) is a free image editing software, **https://fiji.sc/**
- It has a powerful "Script Editor", using a very simple version of "JavaScript" ("IJ1 Macro").
- FIJI is so much more than just image editing/processing – it's a massively underestimated programming platform.



„IJ1 Macro“

- Quick familiarization and quick success.
- Code that can (almost) easily be read by humans.
- You don't have to include any “libraries” (C, Python) and you can get started right away.
- “Auto-completion" inserts whole code parts ("snippets").
- Massive fun factor, few commands.





„ImageJ“ and „FIJI“

- “FIJI” is a version of ImageJ that already contains countless plugins (the focus is on scientific image processing).
- Both are free.
- There are regular updates / patches.
- The software does not need to be installed; portable, runs on computers without any administrator rights.
- At **https://imagej.nih.gov/ij/developer/macro/functions.html** you can find a complete (current) list with a detailed description of every single command - you can go astonishingly far with that.
- It’s like LEGO - complex structures from very simple blocks.

Our concrete example

- We build a prime number generator in ImageJ and also in C++
- Don't worry, it won't be the boring dull “brute force” calculator;
  our generator will be beautiful, elegant and calculates shockingly little.
- You can download all the codes presented and discussed here from

www.github.com/Myocyter/PrimeLister-v1.1

or

www.scyrus.de/PrimeLister.zip

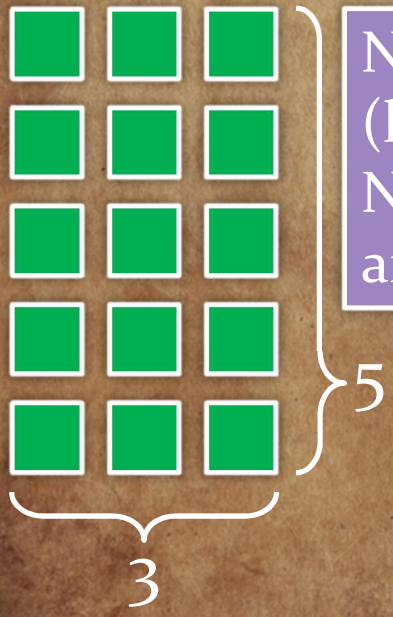
What is a prime number?

A prime number (or prime) is a natural number greater than 1 that is not a product of smaller natural numbers - and therefore only divisible by itself and by 1.

1 is not a prime.

Primes are yellow, non-primes white:

2, **3**, 4, **5**, 6, **7**, 8, 9, 10, **11**, 12, **13**, 14, 15, 16, **17**, 18, **19**, 20, 21, 22, **23**, 24, 25,...



Non-primes like 15 can be „created“ using primes (here: 3 and 5); non-primes are products of primes. No prime can be created that way, primes can‘t be arranged as products of other numbers.

The moronic, boring primality test

Every number N that is not prime can be written as the product of prime numbers that are smaller than N (factorization) – or, to be precise, ≤ $\sqrt{N}$.

15 = 3 x 5 ; sqrt(15)≈3,873 25 = 5 x 5 ; sqrt(25)=5 143 = 11 x 13; sqrt(143)≈11,958

Question: is 101 a prime? sqrt(101) ≈ 10,04; so, we just have to test 2, 3, 5, and 7 – 11 is already too big.

Question: is 15.485.863 a prime? sqrt(15,485,863) ≈ 3935,208.
There are 546 primes smaller than 3935.
15.485.863 is prime, so we actually have to calculate all of these 546 tests.

However, with our generator we want to reach 10+ billions (9,592 primes are smaller than $\sqrt{10^{10}}$).

Therefore, "brute force"-testing is not an option and we will avoid testing at all.

About prime numbers

			Array A		Array B	
			6*n-1		6*n+1	*Element*
2	**3**	4	**5**	6	**7**	***0***
8	9	10	**11**	12	**13**	***1***
14	15	16	**17**	18	**19**	***2***
20	21	22	**23**	24	25	***3***
26	27	28	**29**	30	**31**	***4***
32	33	34	35	36	**37**	***5***
38	39	40	**41**	42	**43**	***6***
44	45	46	**47**	48	49	***7***
50	51	52	**53**	54	55	***8***
56	57	58	**59**	60	**61**	***9***
62	63	64	65	66	**67**	***10***
68	69	70	**71**	72	**73**	***11***
74	75	76	77	78	**79**	***12***
80	81	82	**83**	84	85	***13***
86	87	88	**89**	90	91	***14***
92	93	94	95	96	**97**	***15***
98	99	100	**101**	102	**103**	***16***
104	105	106	**107**	108	**109**	***17***
110	111	112	**113**	114	115	***18***

All prime numbers are either
"(multiples of 6)–1" or
"(multiples of 6)+1".

We just need to look at two columns:
"6*n-1" (Array A)
and
"6*n+1" (Array B)

Not all of these numbers are primes, but ALL of the primes (≥5) are found exclusively here.

About prime numbers

			Array A		Array B	
			6*n-1		6*n+1	*Element*
2	**3**	4	**5**	6	**7**	***0***
8	9	10	**11**	12	**13**	***1***
14	15	16	**17**	18	**19**	***2***
20	21	22	**23**	24	25	***3***
26	27	28	**29**	30	**31**	***4***
32	33	34	35	36	**37**	***5***
38	39	40	**41**	42	**43**	***6***
44	45	46	**47**	48	49	***7***
50	51	52	**53**	54	55	***8***
56	57	58	**59**	60	**61**	***9***
62	63	64	65	66	**67**	***10***
68	69	70	**71**	72	**73**	***11***
74	75	76	77	78	**79**	***12***
80	81	82	**83**	84	85	***13***
86	87	88	**89**	90	91	***14***
92	93	94	95	96	**97**	***15***
98	99	100	**101**	102	**103**	***16***
104	105	106	**107**	108	**109**	***17***
110	111	112	**113**	114	115	***18***

Waste of time

Waste of time

All prime numbers are either
"(multiples of 6)–1" or
"(multiples of 6)+1".

We just need to look at two columns:
"6*n-1" (Array A)
and
"6*n+1" (Array B)

Not all of these numbers are primes, but ALL of the primes (≥5) are found exclusively here.

An Array is just a list of elements. Here, those elements are „prime candidates“. Those Arrays are called

Array A (6*n -1)

and

Array B (6*n+1)

In FIJI the corresponding code looks like this:

```
range=1000000;
A=newArray(range);
B=Array.copy(A);
```

Array A:
A[0], A[1], A[2],…, A[999.999]

```
for (i = 1; i < A.length+1; i++) {
A[i-1]=6*i-1;
B[i-1]=6*i+1;}
```

„A.length“ returns the length of the according Array;

```
range=1000000;
A=newArray(range);
B=Array.copy(A);

for (i = 1; i < A.length+1; i++) {
A[i-1]=6*i-1;
B[i-1]=6*i+1;}
```

Input:

```
print(A[10]);
```

Output:

65

Cool.

But how to get rid of all that boring non-primes?

Array A	
6*n-1	
5	A[0]
11	A[1]
17	A[2]
23	A[3]
29	A[4]
35	A[5]
41	A[6]
47	A[7]
53	A[8]
59	A[9]
65	A[10]
71	A[11]
77	A[12]
83	A[13]
89	A[14]
95	A[15]
101	A[16]
107	A[17]
113	A[18]...

Array B	
6*n+1	
7	B[0]
13	B[1]
19	B[2]
25	B[3]
31	B[4]
37	B[5]
43	B[6]
49	B[7]
55	B[8]
61	B[9]
67	B[10]
73	B[11]
79	B[12]
85	B[13]
91	B[14]
97	B[15]
103	B[16]
109	B[17]
115	B[18]...

Array A									
5	11	17	23	29	35	41	47	53	59
65	71	77	83	89	95	101	107	113	119
125	131	137	143	149	155	161	167	173	179
185	191	197	203	209	215	221	227	233	239
245	251	257	263	269	275	281	287	293	299
305	311	317	323	329	335	341	347	353	359
365	371	377	383	389	395	401	407	413	419
425	431	437	443	449	455	461	467	473	479
485	491	497	503	509	515	521	527	533	539
545	551	557	563	569	575	581	587	593	599
605	611	617	623	629	635	641	647	653	659
665	671	677	683	689	695	701	707	713	719
725	731	737	743	749	755	761	767	773	779
785	791	797	803	809	815	821	827	833	839
845	851	857	863	869	875	881	887	893	899
905	911	917	923	929	935	941	947	953	959

We start with the first element of the Array (A[0]), read its value (here: 5) and, starting from A[0], delete every 5th element (A[5], A[10], A[15], A[20], ...).

Array A
6*n-1
5
11
17
23
29
35
41
47
53
59
65
71
77
83
89
95
101
107
113

A[0]
A[1]
A[2]
A[3]
A[4]
A[5]
A[6]
A[7]
A[8]
A[9]
A[10]
A[11]
A[12]
A[13]
A[14]
A[15]
A[16]
A[17]
A[18]

Array A									
5	11	17	23	29	35	41	47	53	59
65	71	77	83	89	95	101	107	113	119
125	131	137	143	149	155	161	167	173	179
185	191	197	203	209	215	221	227	233	239
245	251	257	263	269	275	281	287	293	299
305	311	317	323	329	335	341	347	353	359
365	371	377	383	389	395	401	407	413	419
425	431	437	443	449	455	461	467	473	479
485	491	497	503	509	515	521	527	533	539
545	551	557	563	569	575	581	587	593	599
605	611	617	623	629	635	641	647	653	659
665	671	677	683	689	695	701	707	713	719
725	731	737	743	749	755	761	767	773	779
785	791	797	803	809	815	821	827	833	839
845	851	857	863	869	875	881	887	893	899
905	911	917	923	929	935	941	947	953	959

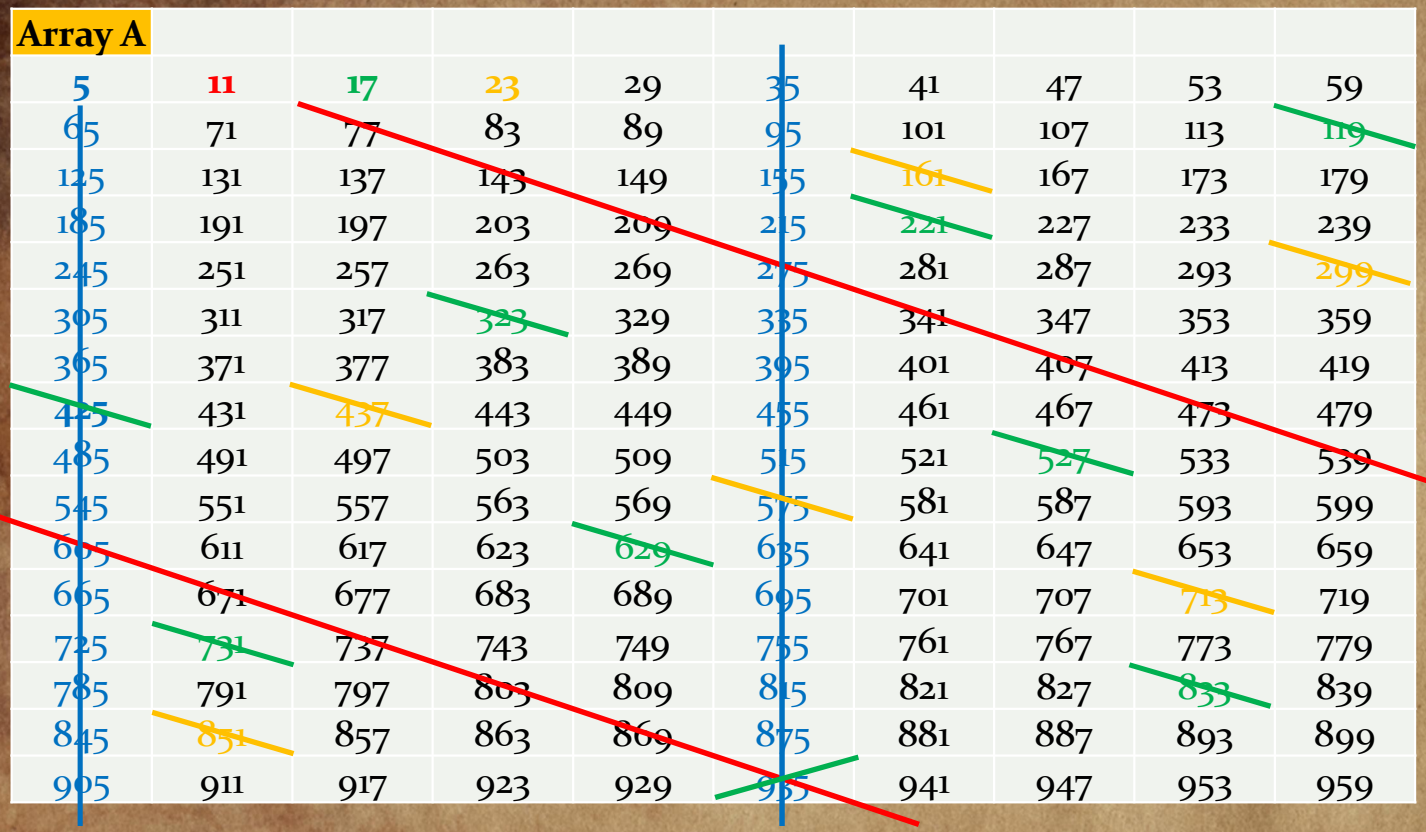
We start with the first element of the Array (A[0]), read its value (here: 5) and, starting from A[0], delete every 5th element (A[5], A[10], A[15], A[20], ...).
All of these elements are multiples of 5 (blue lines).
Here, “delete” means in fact “replace with 0”.

Array A 6*n-1	
5	A[0]
11	A[1]
17	A[2]
23	A[3]
29	A[4]
35	A[5]
41	A[6]
47	A[7]
53	A[8]
59	A[9]
65	A[10]
71	A[11]
77	A[12]
83	A[13]
89	A[14]
95	A[15]
101	A[16]
107	A[17]
113	A[18]

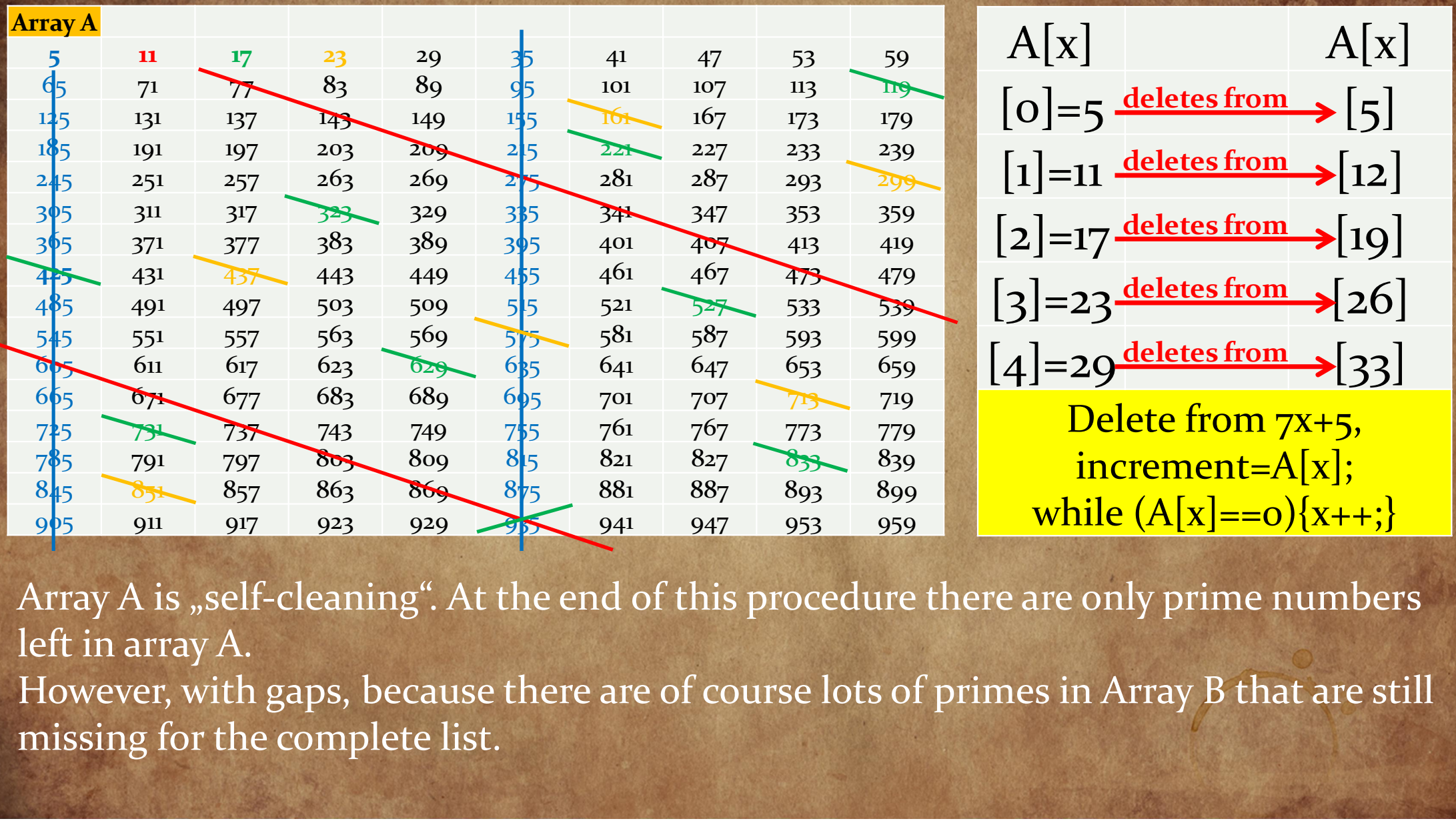
Array A									
5	11	17	23	29	35	41	47	53	59
65	71	77	83	89	95	101	107	113	119
125	131	137	143	149	155	161	167	173	179
185	191	197	203	209	215	221	227	233	239
245	251	257	263	269	275	281	287	293	299
305	311	317	323	329	335	341	347	353	359
365	371	377	383	389	395	401	407	413	419
425	431	437	443	449	455	461	467	473	479
485	491	497	503	509	515	521	527	533	539
545	551	557	563	569	575	581	587	593	599
605	611	617	623	629	635	641	647	653	659
665	671	677	683	689	695	701	707	713	719
725	731	737	743	749	755	761	767	773	779
785	791	797	803	809	815	821	827	833	839
845	851	857	863	869	875	881	887	893	899
905	911	917	923	929	935	941	947	953	959

Then we take the second element of the Array (A[1]), read it out (11) and remove, starting from A[1] every 11th element (A[12], A[23], A[34], A[45], and so on...).
All those elements are multiples of 11 (red lines).

Array A	
6*n-1	
5	A[0]
11	A[1]
17	A[2]
23	A[3]
29	A[4]
35	A[5]
41	A[6]
47	A[7]
53	A[8]
59	A[9]
65	A[10]
71	A[11]
77	A[12]
83	A[13]
89	A[14]
95	A[15]
101	A[16]
107	A[17]
113	A[18]



And this is exactly how it continues, with every single element that has not yet been replaced by 0 / is bigger than 0. We don't have to calculate, we don't have to factorize numbers, we just have to delete - whether the number to be deleted has 2 or 200 digits does not matter. This happens lightning fast in arrays, larger numbers do not cost us more time - the amount of elements to delete determines the duration.



Array A is „self-cleaning“. At the end of this procedure there are only prime numbers left in array A.

However, with gaps, because there are of course lots of primes in Array B that are still missing for the complete list.

With Array B we proceed exactly as with array A. We take element B[0] (=7) and delete its multiples, starting from element B[7] (=49)

A0	5	7	B0
A1	11	13	B1
A2	17	19	B2
A3	23	25	B3
A4	29	31	B4
A5	35	37	B5
A6	41	43	B6
A7	47	49	B7
A8	53	55	B8
A9	59	61	B9
A10	65	67	B10
A11	71	73	B11
A12	77	79	B12
A13	83	85	B13
A14	89	91	B14
A15	95	97	B15
A16	101	103	B16
A17	107	109	B17
A18	113	115	B18
A19	119	121	B19

Then, we delete the multiples of element B[1] (=13), starting from element B[14] (=91), and so on…

Here the "delete from element" formula is not 7x+5 as in Array A, but 7x+7.

Unfortunately, Array B is NOT completely "self-cleaning".
The squares of the prime numbers of Array A are found in Array B.

A0	5	7	B0
A1	11	13	B1
A2	17	19	B2
A3	23	25	B3
A4	29	31	B4
A5	35	37	B5
A6	41	43	B6
A7	47	49	B7
A8	53	55	B8
A9	59	61	B9
A10	65	67	B10
A11	71	73	B11
A12	77	79	B12
A13	83	85	B13
A14	89	91	B14
A15	95	97	B15
A16	101	103	B16
A17	107	109	B17
A18	113	115	B18
A19	119	121	B19

So we need Array A in order to be able to delete several non-prime numbers from Array B - of course the squares of prime numbers are never primes themselves.

A47	287	289	B47
A48	293	295	B48
A49	299	301	B49
A50	305	307	B50

Now the squares of the primes of Array A must be deleted from Array B, just like the multiples of primes from array A, starting from its square:

A0	5	7	B0
A1	11	13	B1
A2	17	19	B2
A3	23	25	B3
A4	29	31	B4
A5	35	37	B5
A6	41	43	B6
A7	47	49	B7
A8	53	55	B8
A9	59	61	B9
A10	65	67	B10
A11	71	73	B11
A12	77	79	B12
A13	83	85	B13
A14	89	91	B14
A15	95	97	B15
A16	101	103	B16
A17	107	109	B17
A18	113	115	B18
A19	119	121	B19

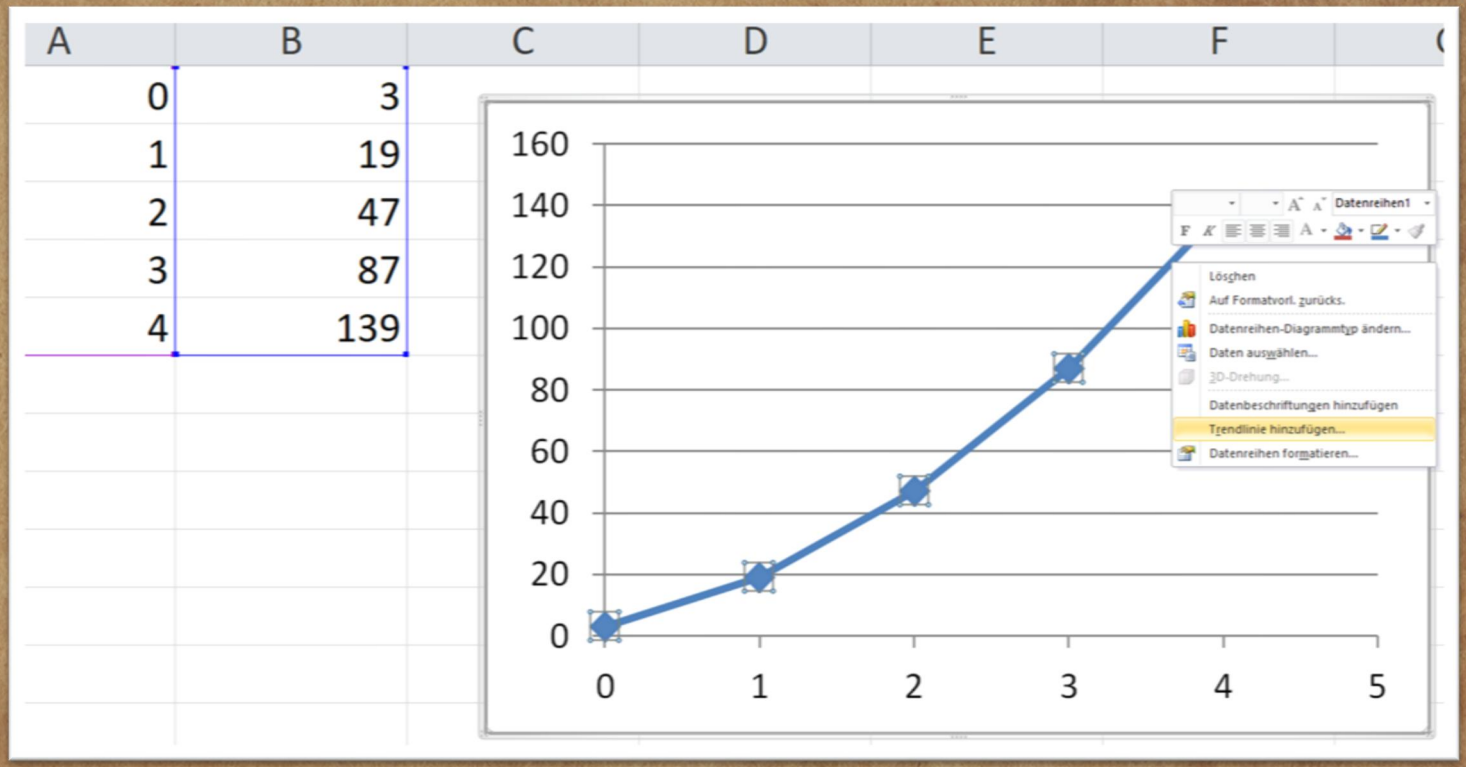
A[0] = 5 deletes from B[3] = 25
A[1] = 11 deletes from B[19] = 121
A[2] = 17 deletes from B[47] = 289
A[3] = 23 deletes from B[87] = 529
A[4] = 29 deletes from B[139] = 841
...

The formula, that brings you from the element in Array A to its square in Array B is:
$6*x^2 + 10*x + 3$

Starting from there, elements in B are deleted using increments of A[x].
A[0]=5, deletes from B[3] in increments of 5;
A[1]=11, deletes from B[19] in increments of 11;
A[2]=17, deletes from B[47] in increments of 17; and so on...

I am not Ramanujan, how should I come up with such a formula? Where is this $6*x^2+10*x+3$ from?
Just list the elements you want to „pair“:

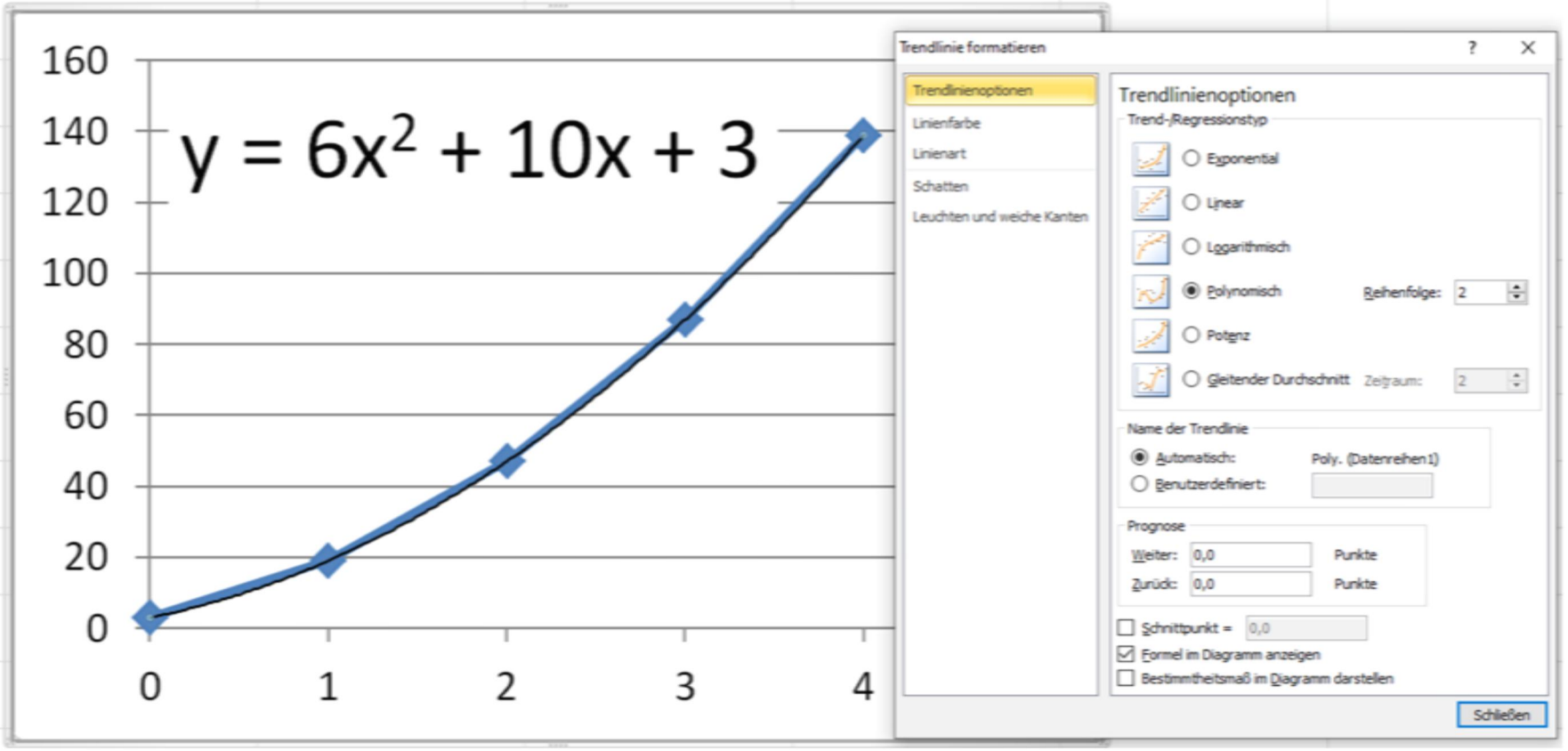
A[0] → B[3]
A[1] → B[19]
A[2] → B[47]
A[3] → B[87]
A[4] → B[139]



If you are not sure, what you are looking for, take Excel (or something similar), plot the curve, add a „trend line“, select „polynomial“ and „show formula in diagram“, there you are…

That‘s perfect to play around quickly and to test ideas.

0	3
1	19
2	47
3	87
4	139



6*x² + 10*x +3 will not be our last „problem“ of this kind.
There are many programs that can fit curves like this.

And that‘s all: In both Arrays A and B all non-primes are already replaced by zero.

How does the according code for ImageJ look like?

```
// Clean Array A of non-primes
// Delete the multiples of every single element in Array A.

for (i = 0; (i+A[i]) < A.length; i++) {
while (A[i]==0){i++;}
for (j = (7*i+5); j < A.length; j=j+A[i]) {A[j]=0;}
}
```

Loop from A[0] to the end of the Array

If the element is already zero, go to the next until you find one >0.

Another loop: delete starting from 7*i+5, use the value of the element A[i] as increment, move on until the end of Array A is reached.
A.length returns the number of elements found in Array A.

All non-primes of Array A are now replaced by zero.

That's what we just did to Array A:

```
for (i = 0; (i+A[i]) < A.length; i++) {
while (A[i]==0){i++;}
for (j = (7*i+5); j < A.length; j=j+A[i]) {A[j]=0;}}
```

Loop from A[0] to the end of the Array

If the element is already zero, go to the next until you find one >0.

Another loop: delete starting from 7*i+5, use the value of the element A[i] as increment, move on until the end of Array A is reached.

Now, we do the same to Array B:

```
for (i = 0; (i+B[i]) < B.length; i++) {
while (B[i]==0){i++;}
for (j = (7*i+7); j < B.length; j=j+B[i]) {B[j]=0;}
}
```

Loop from B[0] to the end of the Array

If the element is already zero, go to the next until you find one >0.

Another loop: delete starting from 7*i+7, use the value of the element B[i] as increment, move on until the end of Array B is reached.

Now we “zero” the squares of the primes from array A and also their multiples in Array B:

```
// Further processing of Array B (slightly more complicated than A)
// Delete all the squares of the elements from array A in Array B
// Delete all the multiples of the elements from Array A in Array B

for (i = 0; (6*i*i+10*i+3) < A.length; i++) {
B[6*i*i+10*i+3]=0;
while (A[i]==0) {i++;}
for (j = (6*i*i+10*i+3); j < B.length; j=j+A[i]) {B[j]=0;}}
```

Start loop at i=0, end if $(6*i^2 + 10*i + 3) \geq$ A.length.

If A[i]=0, it was no prime, so go to the next element >0 (a prime)

Start deleting the elements in Array B at $(6*i^2 + 10*i + 3)$, use A[i] as increment.

Almost done.
Now we have two Arrays (A and B) that contain only primes and zeros.
All that's missing is some cosmetics.

```
A=Array.deleteValue(A, 0);
B=Array.deleteValue(B, 0);
```
} Delete ALL zeros from the Arrays A and B in a single step.

```
A=Array.concat(A,2);
A=Array.concat(A,3);
```
} Insert the two missing primes 2 and 3.

```
A=Array.concat(A,B);
A=Array.sort(A);
```
} Merge both Arrays (Array A becomes A+B) and sort them.

Print the identified primes as table:

```
columns=100;How many columns should the table have? In this case 100.

for (i = 0; i < A.length-colums; i=i+colums) { Loop all elements of A.
line=toString((1+i) + "-" + (i+colums) + ":" + "\t"); Count the elements per row.
for (s = i; s < i+colums; s++) {line=line+A[s]+"\t";} Unite 100 elements to a single row.
print(line); Print that row
}
```

The screen output can either be saved as a text file or using export copy / paste to programs like Excel.

And that's what you get (copied from FIJI to Excel)...

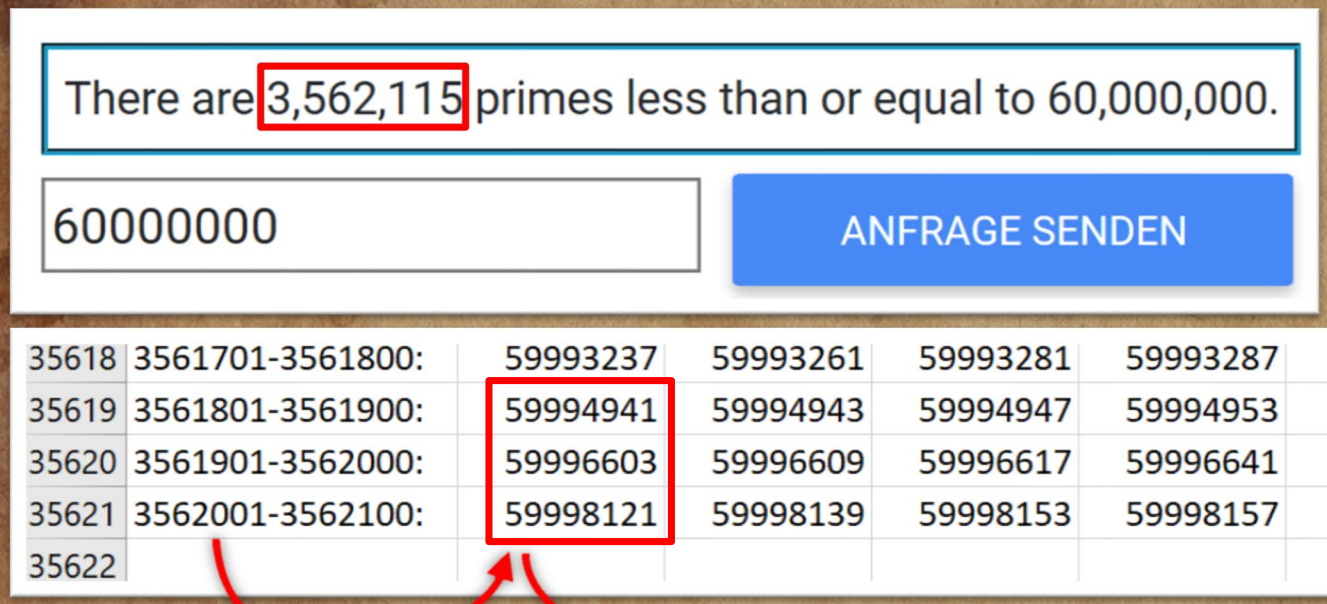
1	1-100	2	3	5	7	11	13	17	19	23
2	101-200	547	557	563	569	571	577	587	593	599
3	201-300	1229	1231	1237	1249	1259	1277	1279	1283	1289
4	301-400	1993	1997	1999	2003	2011	2017	2027	2029	2039
5	401-500	2749	2753	2767	2777	2789	2791	2797	2801	2803

[...]

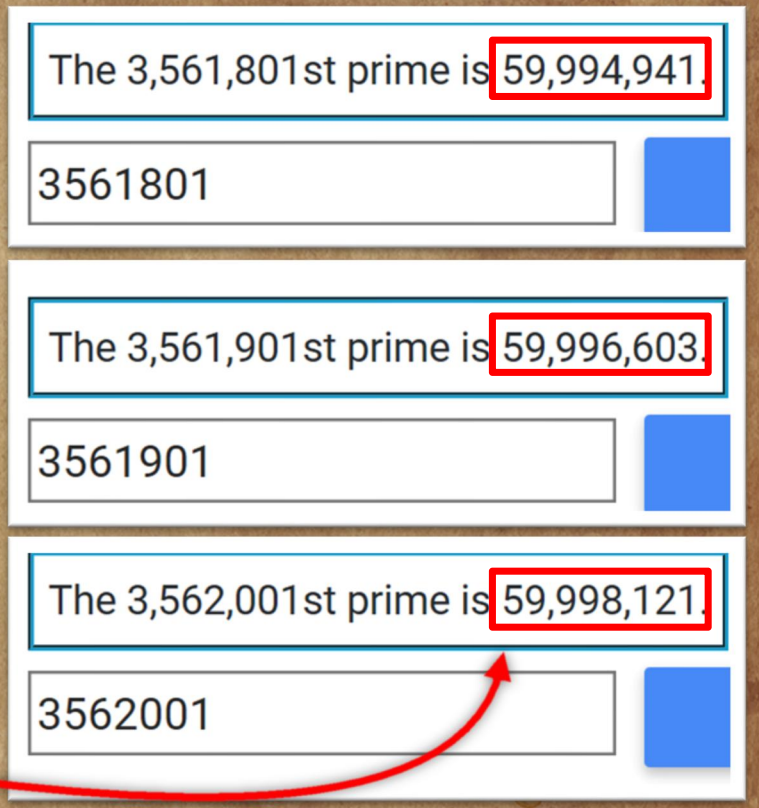
100519	10051801-10051900	180408409	180408443	180408451	180408467	180408469	180408479	180408491	180408493
100520	10051901-10052000	180410179	180410213	180410221	180410233	180410249	180410281	180410287	180410381
100521	10052001-10052100	180412073	180412079	180412081	180412091	180412093	180412097	180412121	180412123
100522	10052101-10052200	180413993	180413999	180414007	180414011	180414043	180414089	180414097	180414119
100523	10052201-10052300	180415967	180415969	180415973	180415981	180415999	180416011	180416029	180416051
100524	10052301-10052400	180417833	180417899	180417953	180417959	180417997	180418001	180418009	180418033
100525									
100526									

But are those primes correct?

https://primes.utm.edu/nthprime/



35618	3561701-3561800:	59993237	59993261	59993281	59993287	599
35619	3561801-3561900:	59994941	59994943	59994947	59994953	599
35620	3561901-3562000:	59996603	59996609	59996617	59996641	599
35621	3562001-3562100:	59998121	59998139	59998153	59998157	599
35622						



Java(Script) and „ImageJ-Java“ are not really known for impressive speed.
Let‘s convert the code to C++.

First a few relevant differences between ImageJ-Script and C++:

ImageJ-Script	C++
Take an array with the according number of elements	Take an empty vector, don‘t use arrays for really large lists of elements.
Assign a number to the already existing elements	Add elements to the existing vector via „vector.push_back (new element); “
Do NOT use a command like „Array.concat(Array, number)“ to add elements to an existing array in a loop => This will become devastatingly slow	Try to avoid sorting of a really large vector via „sort(A.begin(), A.end())“ => Also devastatingly slow

My first attempt in C++ (actually my first program written in C++ ever) failed, because the two vectors A and B (corresponds to the Arrays A and B in ImageJ) had to be merged and sorted.

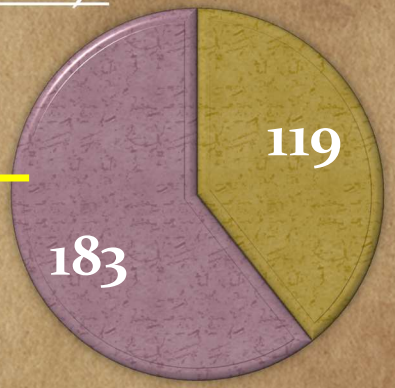
No problem in imageJ, but in C++ it became extremely slow.

Testing the failed C++ version (i7-8700 (3,2 GHz), 32 GB-Ram, Win10 (64-bit));
Compiler: DEV C++, Version 5.11

Calculate all primes ≤ $5*10^9$ (about 235 million primes):

Whole computing time: 302 seconds

From creation of the vectors to deleting all zeros in the merged vectors: 119 seconds



Sorting the primes (**sort(A.begin(), A.end());**): 183 seconds.

Rate without sorting: 1.967.301 primes/sec
Rate with sorting: 778.145 primes/sec

⇨ C++ version is only 3 times faster than ImageJ. Shockingly inadequate.
⇨ We need a new approach, avoid sorting, it must be possible with a single vector.
⇨ Do not transfer the code from one language to the other. Transfer the basic idea.

Recode the idea for C++

```
/ Fill Vector with the Prime-candidates 6n-1 and 6n+1...
for (i = 0; i < bereich; i++) { A.push_back(i * 6 + 5); A.push_back(i * 6 + 7); }

/ Delete all the multiples of the single elements (E0, E2, E4...)
   for (i = 0; (7*i + 10) < A.size(); i=i+2) {
       while (A[i] < 1) {i=i+2;}
       step = 2*A[i];
       for (j = 7*i + 10; j < A.size(); j = j+step) {A[j]=0;}}
```

Start from element 7*i +10; the increment is 2*A[i].

What we did previously in two separate Arrays/Vectors is now done in a single one: First to the elements from Vector A (E0, E2, E4,...) and then to the elements of Vector B (E1, E3, E5,...).

Recode the idea for C++

```
/ Delete all the multiples of the single elements (E1, E3, E5...)

  for (i = 1; (3*i*i + 8*i + 4) < A.size(); i=i+2) {
      while (A[i] < 1) {i=i+2;}
      step = 2*A[i];
      for (j=3*i*i + 8*i + 4; j < A.size(); j=j+step) {A[j]=0;}}
```

Start from element $3*i^2+8*i+4$; the increment is 2*A[i].

What we did previously in two seprate Arrays/Vectors is now done in a single one: First to the elements from Vector A (E0, E2, E4,...) and then to the elements of Vector B (E1, E3, E5,...).

Recode the idea for C++

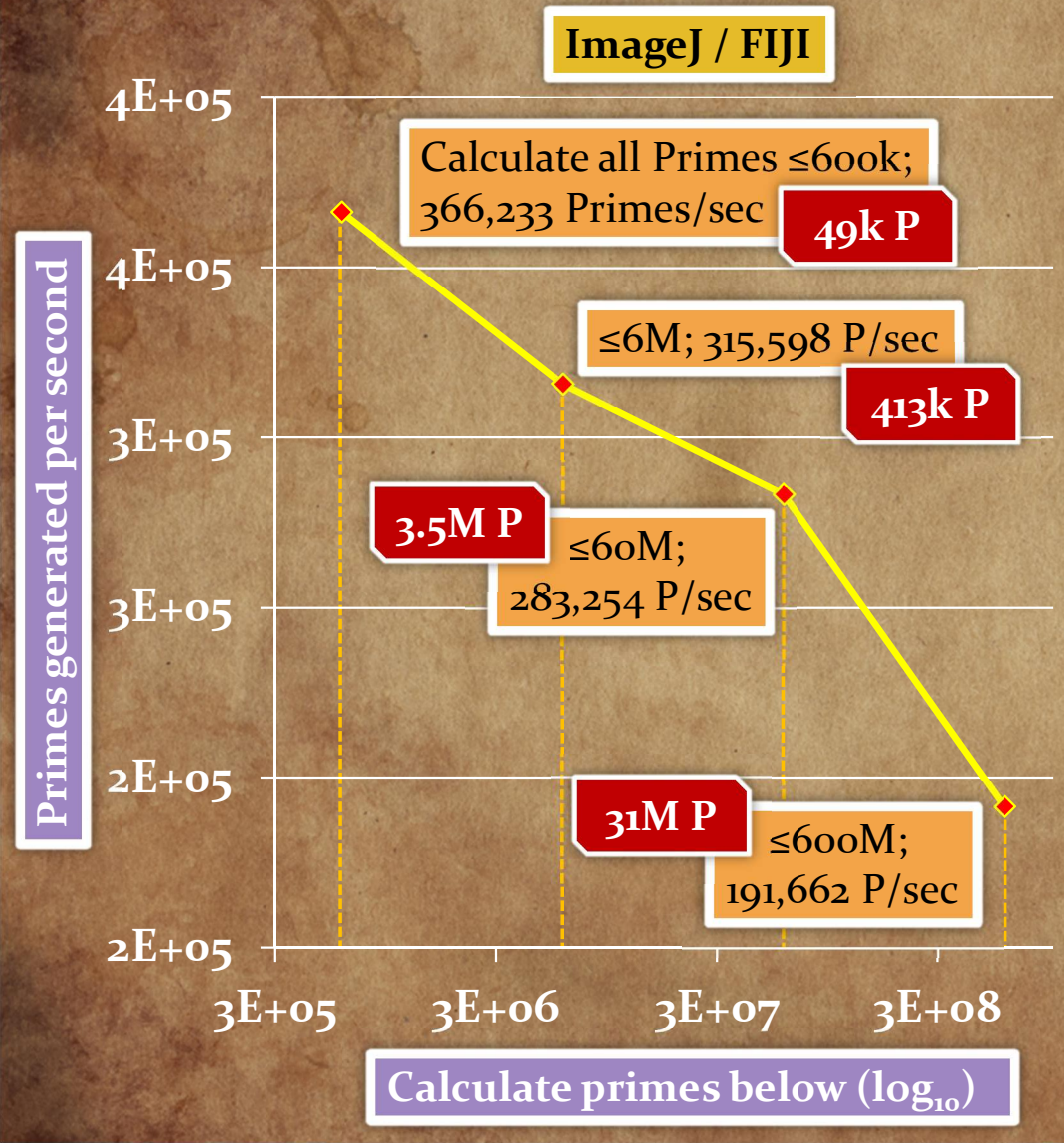
```
/ Now delete all the remaining squares of the single elements (E0, E2, E4...)
for (i = 0; (3*i*i + 10*i+7) < A.size(); i=i+2) {
if (A[3*i*i + 10*i + 7]>0) {step=2*sqrt(A[3*i*i + 10*i + 7]);
A[3*i*i + 10*i + 7]=0;
for (j=(3*i*i + 10*i + 7); j<A.size(); j=j+step){A[j]=0;}}}

/ Erase all the zeros
A.erase(remove(A.begin(), A.end(), 0), A.end());

/ Insert (prepend!) the missing Primes 3 and 2
   A.insert(A.begin(), 3);  A.insert(A.begin(), 2);
```

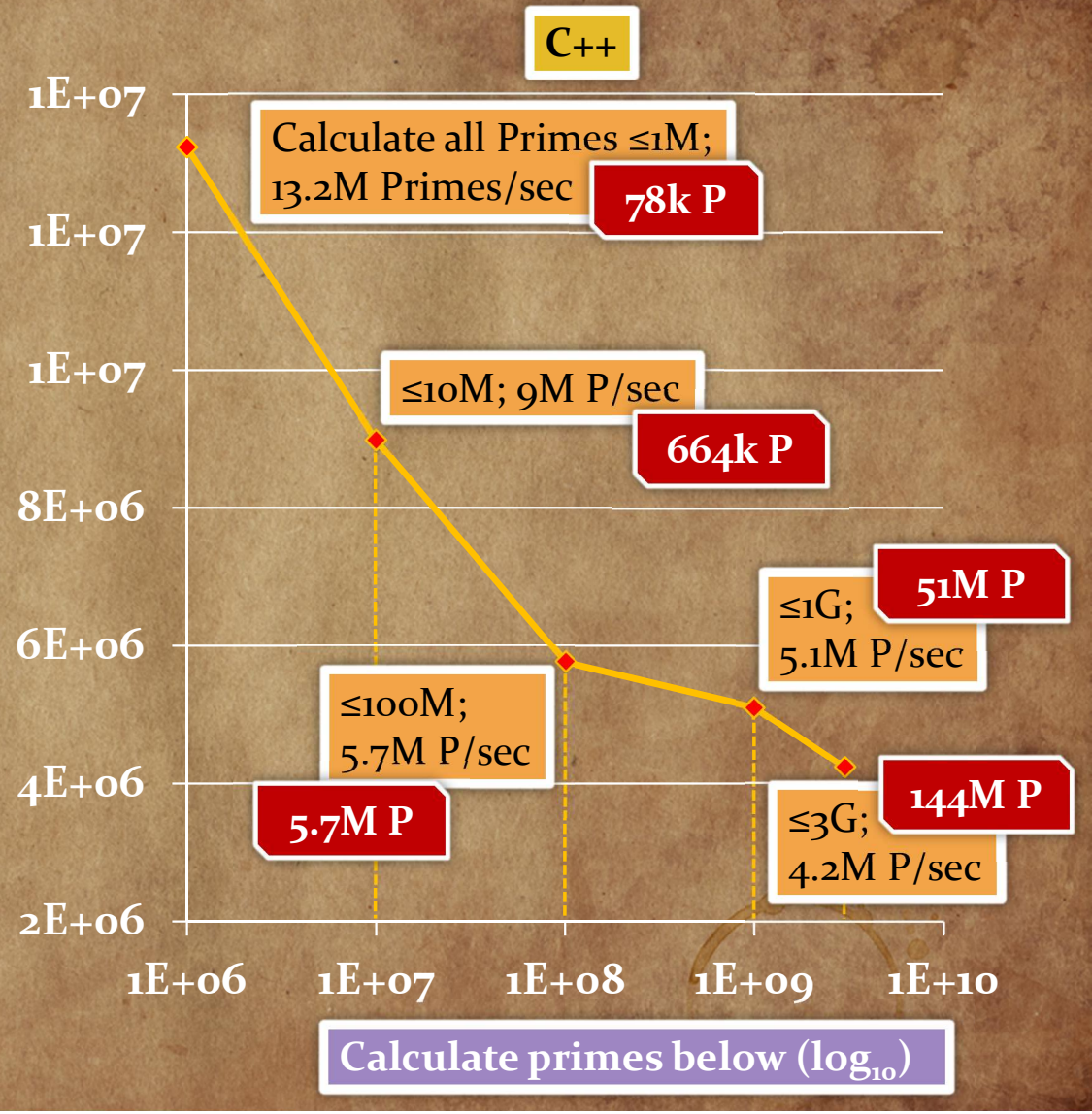
Start from element $3*i^2+10*i+7$

The increment is the root of A[i]

The primes are already in their natural order, no need for sorting.

Speed Test, v1.0 (i7-9750H (2.6 GHz), 16 GB-Ram, Win10 (64-bit)), VS-compiled





But...?

Do we really need the Vector actually filled with prime-candidates?
If necessary, we can calculate the value of any element by its position.

What if we start with a large Vector, assign a „1“ to every single element and start replacing the elements corresponding to non-primes with zeros?

```
// Fill a Vector with ones...

    std::vector<unsigned long long int>A(bereich, 1);
```

“bereich” is the range we want to cover.

Just a last modification of the C++ version…

```
// Delete all the multiples of the single elements (E0, E2, E4…)

   for (i = 0; (7*i+10) < A.size(); i = i+2) {
      while (A[i]==0) { i = i+2; }
      step = 6*i + 10;
      for (j = 7*i + 10; j < A.size(); j=j + step) {A[j] = 0;}}
```

We can calculate “step” from the position of the element via “6*i +10”, there is no need to read the according element.
And then, we just count and replace. First the even…

Just a last modification of the C++ version…

…then the odd elements:

```
// Delete all the multiples of the single elements (E1, E3, E5…)

    for (i = 1; (3*i*i + 8*i + 4) < A.size(); i = i+2) {
        while (A[i] == 0) {i = i+2;}
        step = 6*i + 8;
        for (j = 3*i*i + 8*i + 4; j < A.size(); j = j + step) {A[j]=0;}}
```

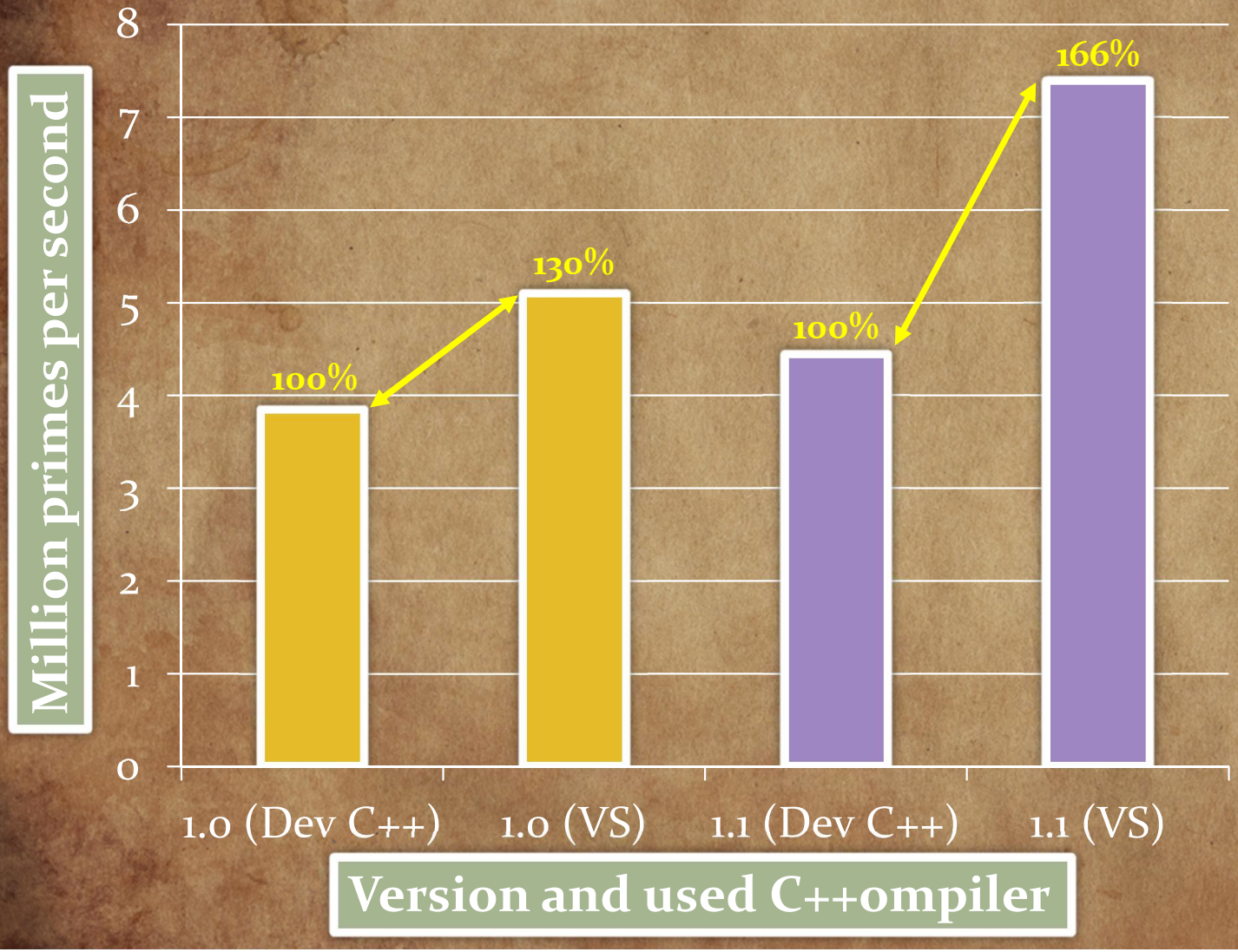
Just a last modification of the C++ version...

And NOW we replace the remaining „1“-elements in the Vector with the according primes:

```
for (i = 0; i < A.size(); i = i+2) {
      while (A[i] == 0 && i < A.size()) {i = i+2;}
      A[i] = 3*i + 5;}


   for (i = 1; i < A.size(); i = i+2) {
      while (A[i] == 0 && i < A.size()) {i = i+2;}
      A[i] = 3*i + 4;}
```

Then, we remove the zeros, add the missing primes 2 and 3. Done.

Time to compare the different (C++) versions...

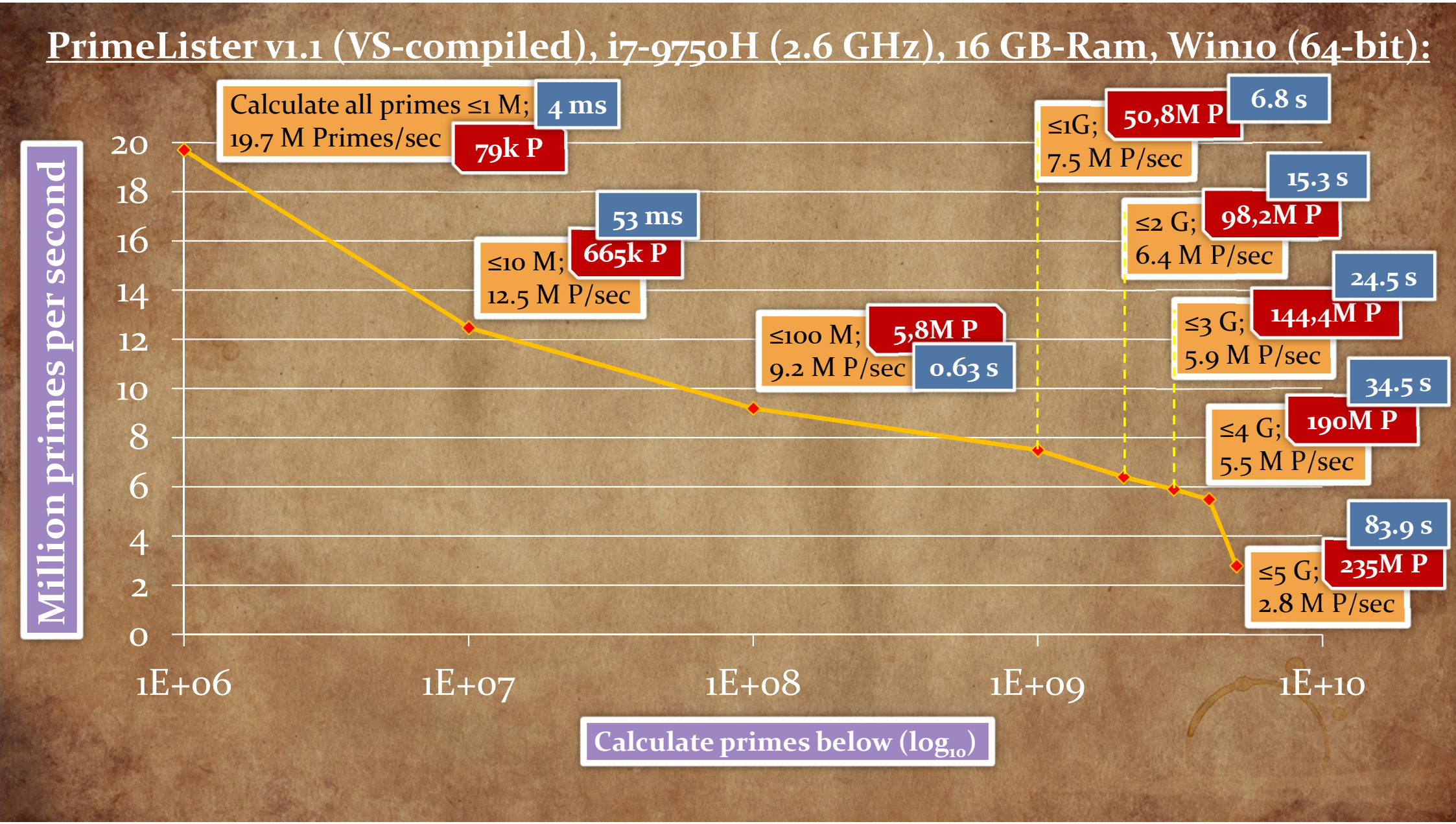


Machine:
i7-9750H (2.6 GHz), 16 GB-Ram, Win10 (64-bit)

Compiler:
VS = Visual Studio 2019
and
Dev C++ (v. 5.11, Bloodshed Software)

Task:
Calculate all primes below 10^9 (about $51*10^6$)

→ The used compiler has massive impact!

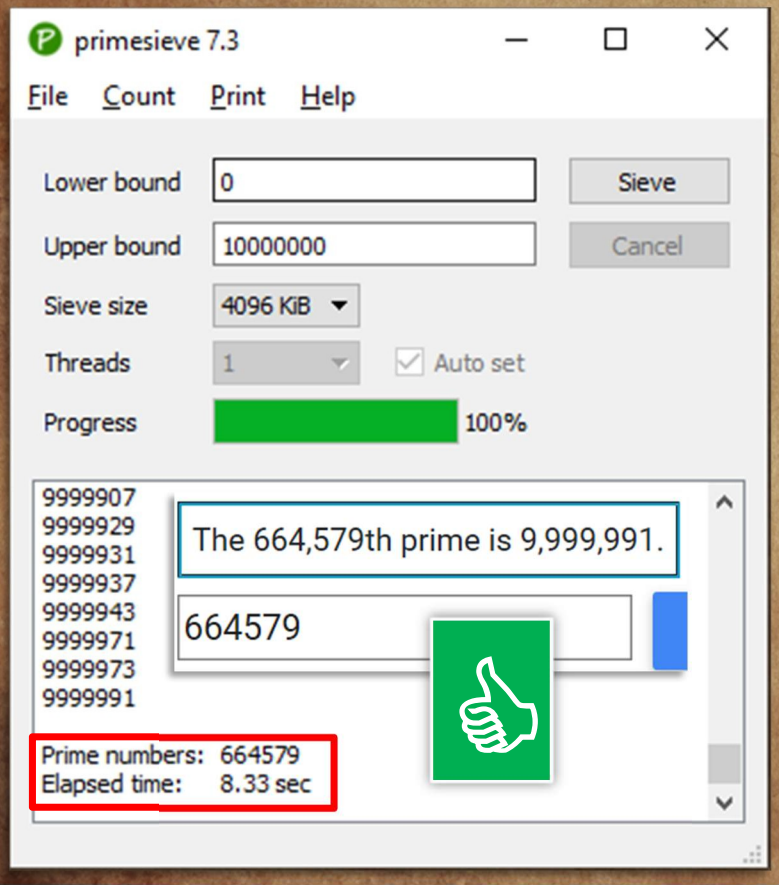

PrimeLister v1.1 (VS-compiled), i7-9750H (2.6 GHz), 16 GB-Ram, Win10 (64-bit):
Million primes per second
20
18
16
14
12
10
8
6
4
2
0
Calculate all primes ≤1 M;
19.7 M Primes/sec
4 ms
79k P
≤10 M;
12.5 M P/sec
53 ms
665k P
≤100 M;
9.2 M P/sec
5,8M P
0.63 s
≤1G;
7.5 M P/sec
50,8M P
6.8 s
≤2 G;
6.4 M P/sec
98,2M P
15.3 s
≤3 G;
5.9 M P/sec
144,4M P
24.5 s
≤4 G;
5.5 M P/sec
190M P
34.5 s
≤5 G;
2.8 M P/sec
235M P
83.9 s
1E+06
1E+07
1E+08
1E+09
1E+10
Calculate primes below (log10)

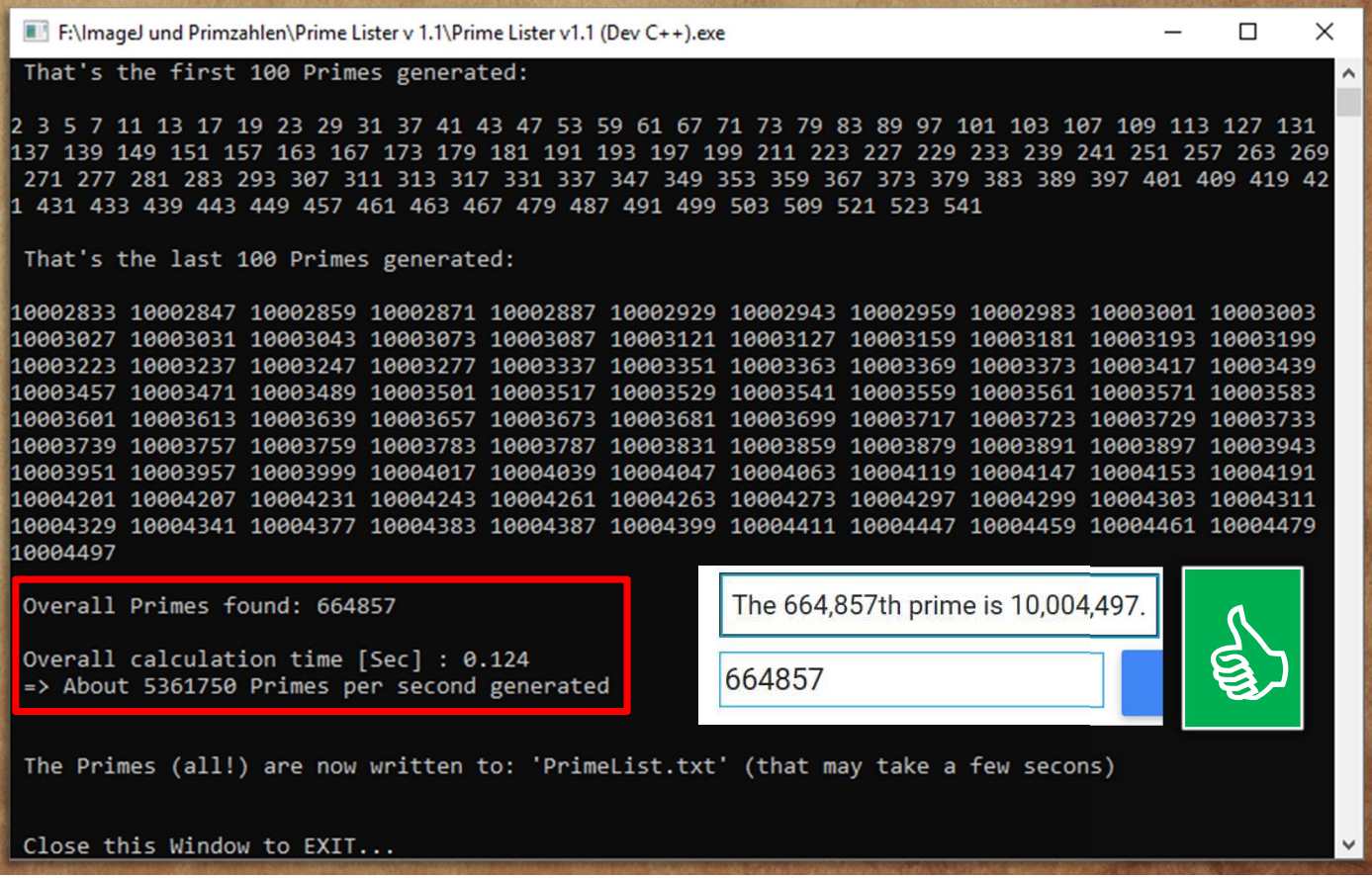
Discussion:

- RAM is the limiting factor.
- The speed of the compiled EXE-file massively depends on the used C++ompiler; compared to „Visual Studio“, the EXE of „Dev C++“ was quite disappointing.
- ImageJ is not really „slow“, but of course it can‘t compete with C++; it has less power and inferior RAM-management.
- Though, reaching up to 19 million primes/sec with C++ and about 360k/sec with FIJI with just a few lines of code is nothing to be ashamed of.
- Identified so far with PrimeLister v1.1: 541,557,779-th prime = 12,000,044,993 (verified via https://primes.utm.edu/nthprime/)

Discussion:

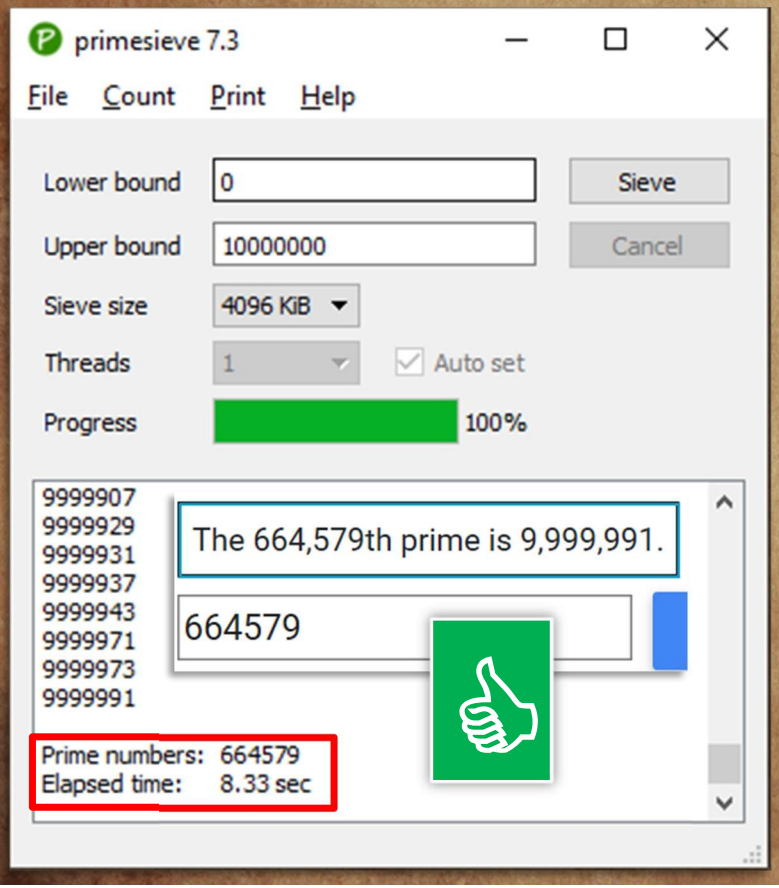
- And compared to others? „primesieve“ is quite fast. Let‘s compare our „Prime Lister v1.1“ to „primesieve v7.3“ **Task:** I need a list of all primes ≤ 10,000,000.





Discussion:

- And compared to others? „primesieve“ is quite fast. Let‘s compare our „Prime Lister v1.1“ to „primesieve v7.3“ **Task:** I need a list of all primes ≤ 10,000,000.



1	Array was created: 246MB of 24324MB (1%)
2	Array filled: 246MB of 24324MB (1%)
3	Delete all the multiples of the single elements (E0, E2, E4,,,)
4	Delete all the multiples of the single elements (E1, E3, E5,,,)
5	Now delete all the remaining squares of the single elements (E0, E2, E4,,,)
6	Range: 3336000
7	Primes identified: 665074
8	Calculation time [sec]: 1,751
9	Primes per second: 379825,243

The 664,901st prime is 10,005,241.

6654	664201-664300:	9993871	9993881	9993887	9993901	9993911	9993913	9993917	9993923	9993941
6655	664301-664400:	9995437	9995477	9995483	9995497	9995519	9995527	9995533	9995549	9995563
6656	664401-664500:	9997093	9997109	9997123	9997129	9997133	9997177	9997189	9997201	9997219
6657	664501-664600:	9998603	9998623	9998633	9998641	9998689	9998699	9998701	9998719	9998741
6658	664601-664700:	10000451	10000453	10000457	10000481	10000511	10000537	10000583	10000591	10000609
6659	664701-664800:	10002007	10002017	10002019	10002029	10002053	10002059	10002061	10002067	10002077
6660	664801-664900:	10003583	10003601	10003613	10003639	10003657	10003673	10003681	10003699	10003717
6661	664901-665000:	10005241	10005251	10005263	10005277	10005293	10005323	10005329	10005337	10005349

If you calculate all primes below 10M in FIJI, calculation takes 1.75 seconds, calculation and printing an overall of 5.5 seconds.

Discussion:

- **Next step:**

  Parallelization of PrimeLister (C++) using #pragma omp;

  tricky in nested loops, first attempts worked, all 6 cores of my processor were used – but computing speed was (slightly) reduced (-5%).

  ⇨ I am working on it.

Downloads:

You get:

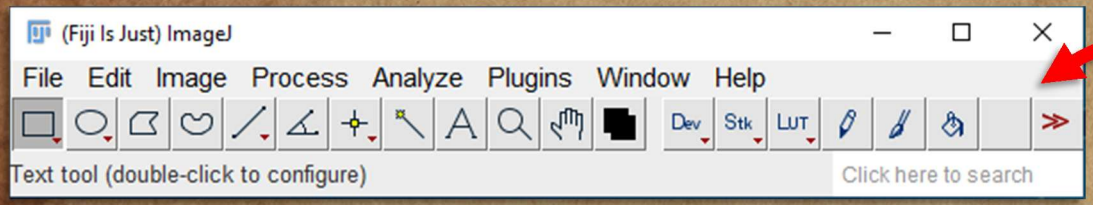
- The source codes for ImageJ („ImageJ PrimeLister (single Array).txt“ and „ImageJ PrimeLister (two Arrays).txt”)
- The source codes for C++ („PrimeLister v1.0 (single vector).cpp“ and „PrimeLister v1.1 (assign primes later).cpp“)
- The EXEs (C++), compiled with both VS and Dev C++ (4 files).

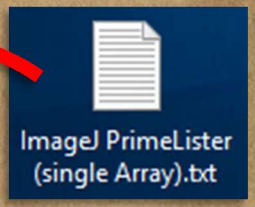
Find everything here:

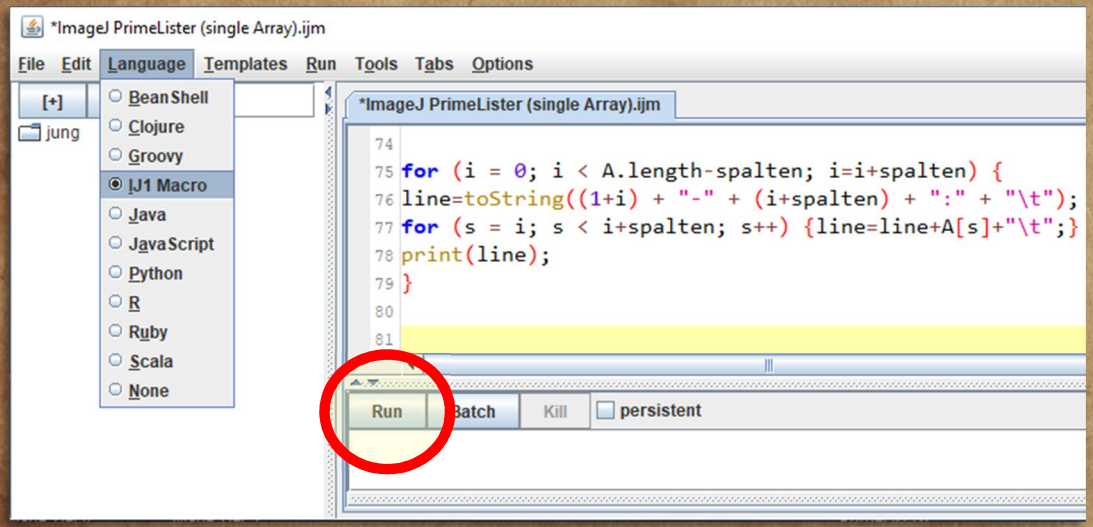
https://github.com/StrangestThings/

How to use the downloads (ImageJ)?

1. Open FIJI, drop the text-file containing the ImageJ-code on the interface







2. Adjust the language to „IJ1 Macro“

3. Edit values for „bereich“ and „spalten“ (if necessary)

4. Click „RUN“