The Sieve of Eratosthenes

Part 2

Genuine versus **Unfaithful Sieve**

The Genuine Sieve of Eratosthenes

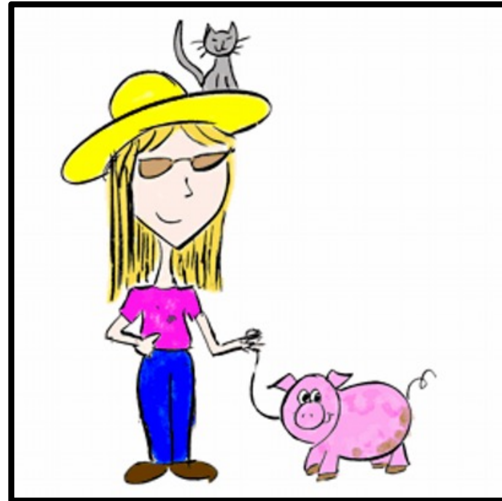
Melissa E. O'Neill

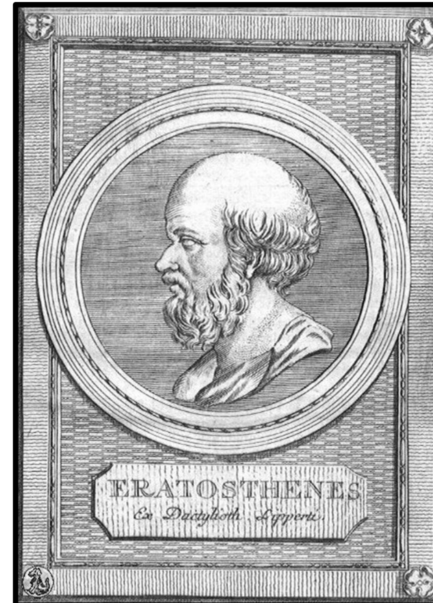
Harvey Mudd College, Claremont, CA, U.S.A. (*e-mail:* `oneill@acm.org`)

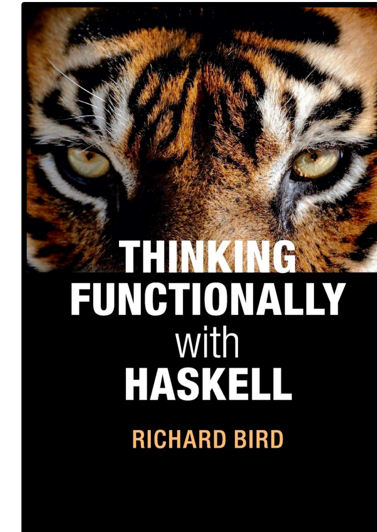
Abstract

A much beloved and widely used example showing the elegance and simplicity of lazy functional programming represents itself as "The Sieve of Eratosthenes". This paper shows that this example is *not* the sieve, and presents an implementation that actually is.

2, 3, 5, 7, 11, ...







Melissa O'Neill

Richard Bird

slides by

https://www.slideshare.net/pjschwarz

When I posted the deck for **Part 1** to the **Scala** users forum, **Odd Möller** linked to the following paper

@philip_schwarz

Related reading:

The Genuine Sieve of Eratosthenes

Melissa E. O'Neill

Harvey Mudd College, Claremont, CA, U.S.A. (e-mail: oneill@acm.org*)*

Abstract

A much beloved and widely used example showing the elegance and simplicity of lazy functional programming represents itself as "The Sieve of Eratosthenes". This paper shows that this example is *not* the sieve, and presents an implementation that actually is.

Odd Möller
@oddan

Scala USERS

The Sieve of Eratosthenes

Announce

philipschwarz May 15

How the effort required to read an understand the Sieve of Eratosthenes varies greatly depending on the programming paradigm used to implement the algorithm.

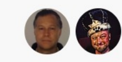
slideshare.net

The Sieve of Eratosthenes - Part 1 8

In this slide deck we are going to see some examples of how the effort required to read an understand the Sieve of Eratosthenes varies greatly depending on the…

2 ♥ Reply

created	last reply	2 replies	146 views	2 users	4 likes	3 links
May 15	May 18					



odd May 16

Related reading: https://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf 6
And here is a gem from the good old days:
Algorithmically challenged: Sieve of Eratosthenes (the real one) Scala One-Liner 9

1 ♥ Reply

philipschwarz May 18

@odd Very interesting - Thank you

The Genuine Sieve of Eratosthenes

1 Introduction

The Sieve of Eratosthenes is a beautiful algorithm that has been cited in introductions to lazy functional programming for more than thirty years (Turner, 1975).

The Haskell code below is fairly typical of what is usually given:

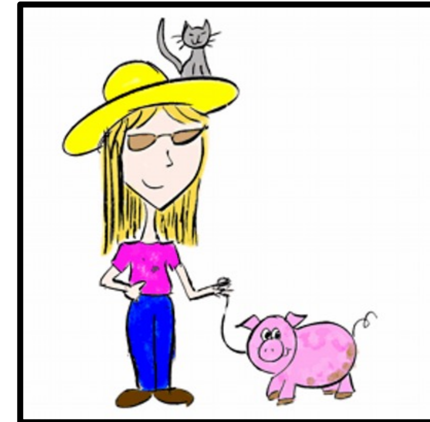
```
primes = sieve [2..]

sieve (p : xs) = p : sieve [x | x <- xs, x `mod` p > 0]
```

The code is short, looks elegant, and seems to make a persuasive case for the power of lazy functional programming.

Unfortunately, on closer inspection, that case begins to fall apart.

For example, the above algorithm actually runs rather slowly, sometimes inspiring excuses as extreme as this one: Try primes !! 19. You should get 71. (This computation may take a few seconds, and do several garbage collections, as there is a lot of recursion going on.) [1]

1 This rather extreme example was found in a spring, 2006, undergraduate programming languages assignment used by several well-respected universities. The original example was not in Haskell (where typical systems require a few orders of magnitude more primes before they bog down), but I have modified it to use Haskell syntax to fit with the rest of this paper.



Melissa O'Neill
@imneme

In the footnote it says that in Haskell, typical systems require a few orders of magnitude more primes before they bog down. On the nexts slide we have a go at timing the primes function and we confirm that it is only when we increase the number of computed primes by between two and three orders of magnitude, i.e. from 10 to between 1,000 and 10,000, that the computation starts taking seconds and using large amounts of memory.

```
> :{
| sieve (p : xs) = p : sieve [x | x <- xs, x `mod` p > 0]
| primes = sieve [2..]
| :}

> :set +s

> primes !! 10
31
(0.00 secs, 388,448 bytes)

> primes !! 100
547
(0.01 secs, 1,571,864 bytes)

> primes !! 1000
7927
(0.22 secs, 131,166,832 bytes)

> primes !! 10000
104743
(20.78 secs, 14,123,155,080 bytes)
```

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,…
(0.00 secs, 405,544 bytes)

> take 100 primes
[2,3,5,7,11,13,17,19,23,…
(0.01 secs, 1,832,824 bytes)

> take 1000 primes
[2,3,5,7,11,13,17,19,23,…
(0.24 secs, 134,539,272 bytes)

> take 10000 primes
[2,3,5,7,11,13,17,19,23,…
(23.97 secs, 14,164,135,832 bytes)
```

The Genuine Sieve of Eratosthenes

2 What the Sieve Is and Is Not

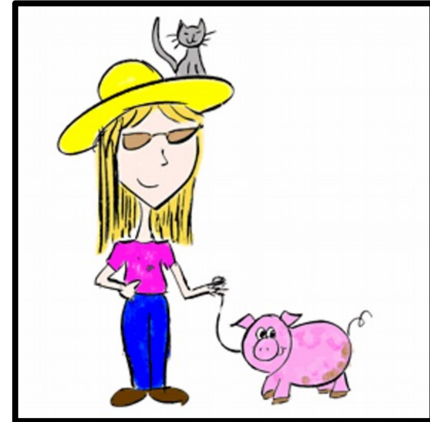
Let us first describe the original "by hand" sieve algorithm as practiced by Eratosthenes.

We start with a table of numbers (e.g., 2, 3, 4, 5, . . .) and progressively cross off numbers in the table until the only numbers left are primes.

Specifically, we begin with the first number, p, in the table, and

1. Declare p to be prime, and cross off all the multiples of that number in the table, starting from p^2;
2. Find the next number in the table after p that is not yet crossed off and set p to that number; and then repeat from step 1.

The starting point of p^2 is a pleasing but minor optimization, which can be made because lower multiples will have already been crossed off when we found the primes prior to p. For a fixed-size table of size n, once we have reached the $\sqrt{n}^{th}$ entry in the table, we need perform no more crossings off—we can simply read the remaining table entries and know them all to be prime. (This optimization does not affect the time complexity of the sieve, however, so its absence from the code in Section 1 is not our cause for worry.)



Melissa O'Neill

In the next 11 slides, we are going to illustrate how the Sieve of Eratosthenes computes the first 100 primes.

Here are the first 100 **prime numbers**, highlighted with a **yellow background**. I am highlighting them from the very beginning to help illustrate that the **Sieve of Eratosthenes** is all about crossing off the **non-prime numbers**, i.e. those with a white background. The first number has a grey background because it is not used by the algorithm.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The first prime number is 2, so we cross off all its multiples, which we highlight with the colour

@philip_schwarz

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The second prime number is 3, so we need to cross off all of its multiples, but some of them, e.g. 6 and 18, have already been crossed off as multiples of two, so to keep this illustration sane, we cross off the uncrossed ones, highlighting them with the colour

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The third prime number is 5, so we cross off all its (uncrossed) multiples, which we highlight with the colour

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The fourth prime number is 7, so we cross off all its (uncrossed) multiples, which we highlight with the colour

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The fifth prime number is 11, so we cross off all its (uncrossed) multiples, which we highlight with the colour

@philip_schwarz

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The sixth prime number is 13, so we cross off all its (uncrossed) multiples, which we highlight with the colour

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The seventh prime number is 17, so we cross off all its (uncrossed) multiples, which we highlight with the colour

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The eigth prime number is 19, so we cross off all its (uncrossed) multiples, which we highlight with the colour

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The ninth prime number is 23, so we cross off all its (uncrossed) multiples, which we highlight with the colour

@philip_schwarz

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The tenth prime number is 29 , so we cross off all its (uncrossed) multiples, which we highlight with the colour .

And we are done: every non-prime number has now been crossed off, so we are left with the first 100 prime numbers (highlighted with a yellow background).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The Genuine Sieve of Eratosthenes

The details of what gets crossed off, when, and how many times, are key to the efficiency of Eratosthenes algorithm.

For example, suppose that we are finding the first 100 primes (i.e., 2 through 541), and have just discovered that 17 is prime, and need to "cross off all the multiples of 17".

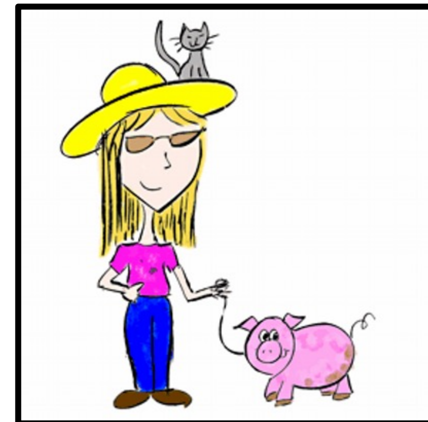
Let us examine how Eratosthenes's algorithm would do so, and then how the algorithm from Section 1 would do so.

In Eratosthenes's algorithm, we start crossing off multiples of 17 at 289 (i.e., 17 × 17) and cross off the multiples 289, 306, 323,..., 510, 527, making fifteen crossings off in total.

Notice that we cross off 306 (17 × 18), even though it is a multiple of both 2 and 3 and has thus already been crossed off twice.[2]

The algorithm is efficient because each composite number, c, gets crossed off f times, where f is the number of unique factors of c less than $\sqrt{c}$.

The average value for f increases slowly, being less than 3 for the first 1012 composites, and less than 4 for the first 1034. [3]



Melissa O'Neill

@imneme

Here (underlined in red) are the 15 multiples of 17 crossed off by the **Sieve of Eratosthenes**. The **crossing off** begins at 17 * 17 = 289 and ends at 527, which is the last multiple of 17 in the table.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541									

The Genuine Sieve of Eratosthenes

Contrast the above behavior with that of the algorithm from Section 1, which I shall call "the unfaithful sieve".

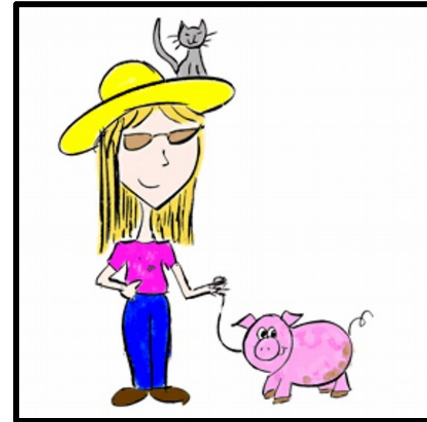
After finding that 17 is prime, the unfaithful sieve will check all the numbers not divisible by 2, 3, 5, 7, 11 or 13 for divisibility by 17.

It will perform this test on a total of ninety-nine numbers (19, 23, 29, 31,..., 523, 527).

The difference between the two algorithms is not merely that the unfaithful sieve doesn't perform "optimizations", such as starting at the square of the prime, or that it uses a divisibility check rather than using a simple increment.

For example, even if it did (somehow) begin at 289, it would still check all forty-five numbers that are not multiples of the primes prior to 17 for divisibility by 17 (i.e., 289, 293, 307,..., 523, 527).

At a fundamental level, these two algorithms "cross off all the multiples of 17" differently.



Melissa O'Neill

```haskell
primes = sieve [2..]

sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p > 0]
```

@philip_schwarz

Here (underlined in red) are the 99 numbers that the **unfaithful sieve** checks for divisibility by 17, beginning with 19 and ending with 527.

Actually, I have underlined 101 numbers, because it seems to me that the **unfaithful sieve** code checks 101 numbers rather than 99, i.e. it also checks 529 and 541, because it doesn't take into account the fact that 527 is the last multiple of 17 contained in the table.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	**19**	20	21	22	**23**	24	25
26	27	28	**29**	30	**31**	32	33	34	35	36	**37**	38	39	40	**41**	42	**43**	44	45	46	**47**	48	49	50
51	52	**53**	54	55	56	57	58	**59**	60	**61**	62	63	64	65	66	**67**	68	69	70	**71**	72	**73**	74	75
76	77	78	**79**	80	81	82	**83**	84	85	86	87	88	**89**	90	91	92	93	94	95	96	**97**	98	99	100
101	102	**103**	104	105	106	**107**	108	**109**	110	111	112	**113**	114	115	116	117	118	119	120	121	122	123	124	125
126	**127**	128	129	130	**131**	132	133	134	135	136	**137**	138	**139**	140	141	142	143	144	145	146	147	148	**149**	150
151	152	153	154	155	156	**157**	158	159	160	161	162	**163**	164	165	166	**167**	168	169	170	171	172	**173**	174	175
176	177	178	**179**	180	**181**	182	183	184	185	186	187	188	189	190	**191**	192	**193**	194	195	196	**197**	198	**199**	200
201	202	203	204	205	206	207	208	209	210	**211**	212	213	214	215	216	217	218	219	220	221	222	**223**	224	225
226	**227**	228	**229**	230	231	232	**233**	234	235	236	237	238	**239**	240	**241**	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	**257**	258	259	260	261	262	**263**	264	265	266	267	268	**269**	270	**271**	272	273	274	275
276	**277**	278	279	280	**281**	282	**283**	284	285	286	287	288	**289**	290	291	292	**293**	294	295	296	297	298	299	300
301	302	303	304	305	306	**307**	308	309	310	**311**	312	**313**	314	315	316	**317**	318	319	320	321	322	**323**	324	325
326	327	328	329	330	**331**	332	333	334	335	336	**337**	338	339	340	341	342	343	344	345	346	**347**	348	**349**	350
351	352	**353**	354	355	356	357	358	**359**	360	**361**	362	363	364	365	366	**367**	368	369	370	371	372	**373**	374	375
376	377	378	**379**	380	381	382	**383**	384	385	386	387	388	**389**	390	**391**	392	393	394	395	396	**397**	398	399	400
401	402	403	404	405	406	407	408	**409**	410	411	412	413	414	415	416	417	418	**419**	420	**421**	422	423	424	425
426	427	428	429	430	**431**	432	**433**	434	435	436	**437**	438	**439**	440	441	442	**443**	444	445	446	447	448	**449**	450
451	452	453	454	455	456	**457**	458	459	460	**461**	462	**463**	464	465	466	**467**	468	469	470	471	472	473	474	475
476	477	478	**479**	480	481	482	483	484	485	486	**487**	488	489	490	**491**	492	**493**	494	495	496	497	498	**499**	500
501	502	**503**	504	505	506	507	508	**509**	510	511	512	513	514	515	516	517	518	519	520	**521**	522	**523**	524	525
526	**527**	528	**529**	530	531	532	533	534	535	536	537	538	539	540	**541**									

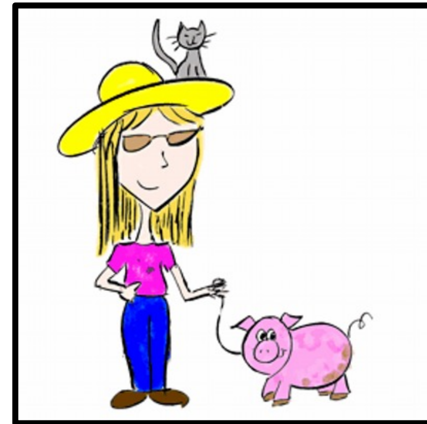
The Genuine Sieve of Eratosthenes

In general, the speed of the unfaithful sieve depends on the number of primes it tries that are not factors of each number it examines, whereas the speed of Eratosthenes's algorithm depends on the number of (unique) primes that are.

We will discuss how this difference impacts their time complexity in the next section. Some readers may feel that despite all of these concerns, the earlier algorithm is somehow "morally" the Sieve of Eratosthenes.

I would argue, however, that they are confusing a mathematical abstraction drawn from the Sieve of Eratosthenes with the actual algorithm.

The algorithmic details, such as how you remove all the multiples of 17, matter.



Melissa O'Neill

It turns out that the **sieve function** from **Part 1** is exactly the **unfaithful sieve**.
In the next three slides we see how the former can be refactored to the latter.

Let's take the code from **Part 1** and do the following:

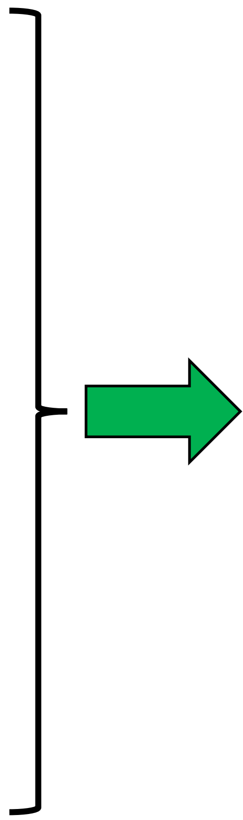
- simplify the **generatePrimes** function by exploiting the behaviour shown on the right
- inline the **divisibleBy** function
- switch to using **mod** in infix mode

```
> [2..1]
[]
> [2..0]
[]
> [2..(-1)]
[]
```

```haskell
generatePrimes :: Int -> [Int]
generatePrimes maxValue =
  if maxValue < 2
  then []
  else sieve [2..maxValue]
```

```haskell
sieve :: [Int] -> [Int]
sieve [] = []
sieve (nextPrime:candidates) =
  nextPrime : sieve noFactors
  where noFactors = filter (not . (`divisibleBy` nextPrime))
                           candidates
```

```haskell
divisibleBy :: Int -> Int -> Bool
divisibleBy x y = mod x y == 0
```



```haskell
generatePrimes :: Int -> [Int]
generatePrimes maxValue = sieve [2..maxValue]
```

```haskell
sieve :: [Int] -> [Int]
sieve [] = []
sieve (nextPrime:candidates) =
  nextPrime : sieve noFactors
  where noFactors = filter (\x -> x `mod` nextPrime > 0)
                           candidates
```

```
haskell> generatePrimes 30
[2,3,5,7,11,13,17,19,23,29]
```

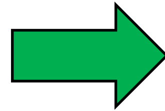
Now let's do the following:

- rename the **generatePrimes** function to **sieve**
- get both functions to deal with an **infinite list**, rather than a **finite length** one
- rewrite the invocation of **filter** as a **list comprehension**

```
> filter (\x -> x `mod` 2 > 0) [1..6]
[1,3,5]

> [x | x <- [1..6], x `mod` 2 > 0]
[1,3,5]
```

```
generatePrimes maxValue = sieve [2..maxValue]
```

```
sieve [] = []
sieve (nextPrime:candidates) =
  nextPrime : sieve noFactors
  where noFactors = filter (\x -> x `mod` nextPrime > 0)
                           candidates
```



```
primes = sieve [2..]
```

```
sieve (nextPrime:candidates) =
  nextPrime : sieve noFactors
  where noFactors = [x | x <- candidates, x `mod` nextPrime > 0)]
```

```haskell
primes = sieve [2..]
```

```haskell
sieve (nextPrime:candidates) =
  nextPrime : sieve noFactors
  where noFactors = [x | x <- candidates, x `mod` nextPrime > 0)]
```

Now let's inline **noFactors**.

```haskell
primes = sieve [2..]
```

```haskell
sieve (nextPrime:candidates) =
  nextPrime : sieve [x | x <- candidates, x `mod` nextPrime > 0)]
```

And finally, let's rename **nextPrime** and **candidates**.

```haskell
primes = sieve [2..]
```

```haskell
sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p > 0)]
```

What we are left with is exactly the **unfaithful sieve**.

The Genuine Sieve of Eratosthenes

If this algorithm is not the Sieve of Eratosthenes, what is it? In fact it is a simple naive algorithm, known as trial division, that checks the primality of x by testing its divisibility by each of the primes less than x. But even this naive algorithm would normally be more efficient, because we would typically check only the primes up to √x. We can write trial division more clearly as

```
primes = 2 : [x | x <-[3..], isprime x]
  isprime x = all (\p -> x `mod` p > 0) (factorsToTry x)
  where
    factorsToTry x = takeWhile (\p -> p*p <= x) primes
```

To futher convince ourselves that we are are not looking at the same algorithm, and to further understand why it matters, it is useful to look at the time performance of the algorithms we have examined so far, both in theory and in practice. For asymptotic time performance, we will examine the time it takes to find all the primes less than or equal to n.

The Sieve of Eratosthenes implemented in the usual way requires $\Theta(n \log \log n)$ operations to find all the primes up to n.

…

Let us now turn our attention to trial division. …

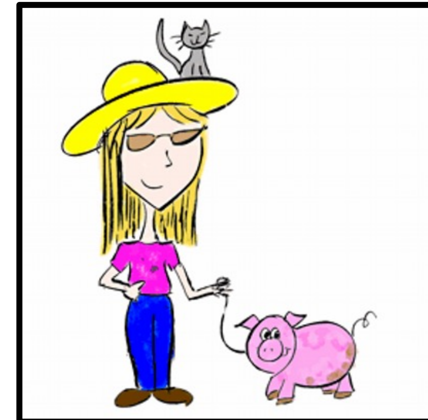
…

From …, we can conclude that trial division has time complexity $\Theta(n \sqrt{n}/(\log n)^2)$.

…

The unfaithful sieve does the same amount of work on the composites as normal trial division …, but it tries to divide primes by all prior primes… and thus the unfaithful sieve has time complexity $\Theta(n^2/(\log n)^2)$.

Thus, we can see that from a time-complexity standpoint, the unfaithful sieve is asymptotically worse than simple trial division, and that in turn is asymptotically worse than than the true Sieve of Eratosthenes.

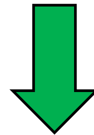


Melissa O'Neill

Let’s translate the **unfaithful sieve** from **Haskell** into **Scala**.

Because the **Haskell** version uses an **infinite list**, in **Scala** we use an **infinite lazy list**.

```
primes = sieve [2..]

sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p > 0]
```



```
def primes = sieve(LazyList.from(2))

def sieve : LazyList[Int] => LazyList[Int] =
  case p #:: xs => p #:: sieve { for x <- xs if x % p > 0 yield x }
```

```scala
def primes = sieve(LazyList.from(2))

def sieve : LazyList[Int] => LazyList[Int] =
  case p #:: xs => p #:: sieve { for x <- xs if x % p > 0 yield x }
```

```
scala> eval(primes.take(100).toList)
val res5: (List[Int], concurrent.duration.Duration) = (List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139,
149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,
257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373,
379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491,
499, 503, 509, 521, 523, 541),3 milliseconds)

scala> eval(primes.take(1_000).toList)(1)
val res1: concurrent.duration.Duration = 54 milliseconds

scala> eval(primes.take(2_000).toList)(1)
val res2: concurrent.duration.Duration = 188 milliseconds

scala> eval(primes.take(3_000).toList)(1)
val res3: concurrent.duration.Duration = 427 milliseconds

scala> eval(primes.take(4_000).toList)(1)
Exception in thread "main" java.lang.StackOverflowError
…
…
```

```scala
def eval[A](expression: => A): (A, Duration) =
  def getTime = System.currentTimeMillis()
  val startTime = getTime
  val result = expression
  val endTime = getTime
  val duration = endTime - startTime
  (result, Duration(duration,"ms"))
```

```scala
def primes = sieve(LazyList.from(2))

def sieve : LazyList[Int] => LazyList[Int] =
  case p #:: xs => p #:: sieve { for x <- xs if x % p > 0 yield x }
```

Now let's change the code so that it works with an ordinary, **finite list**.

```scala
def primes(n: Int): List[Int] = sieve(List.range(2,n+1))

def sieve : List[Int] => List[Int] =
  case Nil => Nil
  case p :: xs => p :: sieve { for x <- xs if x % p > 0 yield x }
```

```scala
def primes(n: Int): List[Int] = sieve(List.range(2,n+1))

def sieve : List[Int] => List[Int] =
  case Nil => Nil
  case p :: xs => p :: sieve { for x <- xs if x % p > 0 yield x }
```

```
scala> eval(primes(541))
val res18: (List[Int], concurrent.duration.Duration) = (List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139,
149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,
257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373,
379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491,
499, 503, 509, 521, 523, 541),0 milliseconds)

scala> eval(primes(1_000))(1)
val res19: concurrent.duration.Duration = 0 milliseconds

scala> eval(primes(10_000))(1)
val res20: concurrent.duration.Duration = 18 milliseconds

scala> eval(primes(20_000))(1)
val res21: concurrent.duration.Duration = 77 milliseconds

scala> eval(primes(50_000))(1)
val res22: concurrent.duration.Duration = 253 milliseconds

scala> eval(primes(100_000))(1)
val res23: concurrent.duration.Duration = 816 milliseconds
```

The Genuine Sieve of Eratosthenes

3 An Incremental Functional Sieve

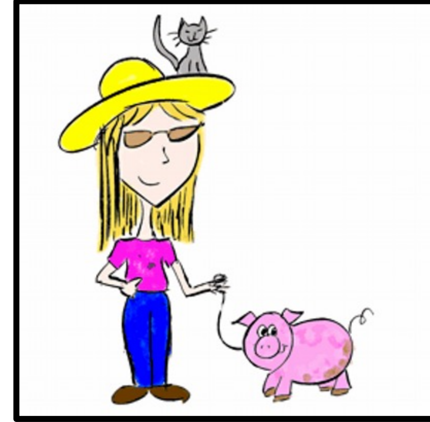
Despite their other drawbacks, the implementations of the unfaithful sieve and trial division that we have discussed use functional data structures and produce an infinite list of primes.

In contrast, classic imperative implementations of the Sieve of Eratosthenes use an array and find primes up to some fixed limit.

Can the genuine Sieve of Eratosthenes also be implemented efficiently and elegantly in a purely functional language and produce an infinite list?

Yes!

Whereas the original algorithm crosses off all multiples of a prime at once, we perform these "crossings off" in a lazier way: crossing off just-in-time…



Melissa O'Neill
@imneme

The rest of section 3, which is the heart of the paper, looks at a number of **'faithful' algorithms**, which rather than using **lists**, use **alternative data structures**, e.g. a **heap**.

Our objective in this deck is much less ambitious than to cover such algorithms.

Instead, what we are going to do next is answer the following question:

> Is it possible to implement a genuine Sieve of Eratosthenes using only lists?

To answer that question we turn to **Richard Bird**'s book: **Thinking Functionally with Haskell** (**TFWH**).

Actually, before moving on to **Richard Bird**'s book, let's have a quick look at the conclusion of the paper.

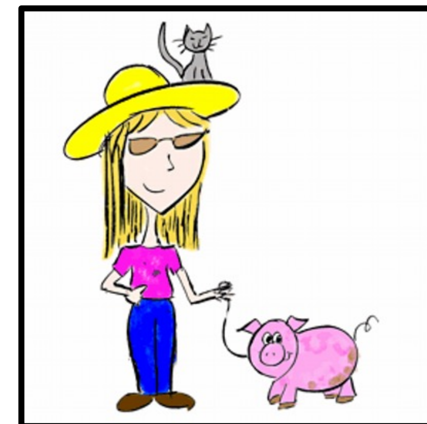
The Genuine Sieve of Eratosthenes

4 Conclusion

A "one liner" to find a lazy list of prime numbers is a compelling example of the power of laziness and the brevity that can be achieved with the powerful abstractions present in functional languages. But, despite fooling some of us for years, the algorithm we began with isn't the real sieve, nor is it even the most efficient one liner that we can write.

An implementation of the actual sieve has its own elegance, showing the utility of well-known data structures over the simplicity of lists. It also provides a compelling example of why data structures such as heaps exist even when other data structures have similar $O(\log n)$ time complexity—choosing the right data structure for the problem at hand made an order of magnitude performance difference.

The unfaithful-sieve algorithm does have a place as an example. It is very short, and it also serves as a good example of how elegance and simplicity can beguile us. Although the name The Unfaithful Sieve has a certain ring to it, given that the unfaithful algorithm is nearly a thousand times slower than our final version of the real thing to find about 5000 primes, we should perhaps call it The Sleight on Eratosthenes.



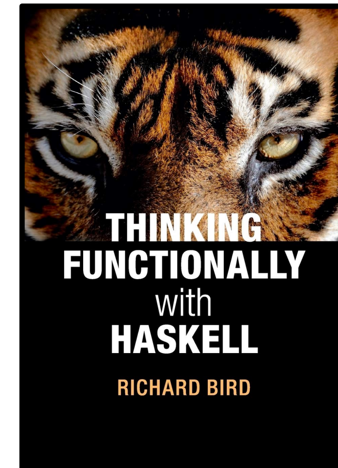
Melissa O'Neill

The first program for computing **primes** that we come across in **TFWH** is similar to the **unfaithful sieve** (shown on the right), in that it uses **trial division**.

```
primes = sieve [2..]

sieve (p : xs) = p : sieve [x | x <- xs, x `mod` p > 0]
```

Richard Bird

```
primes = [x | x <- [2..], divisors x == [x]]

divisors x = [d | d <- [2..x], x `mod` d == 0]
```



@philip_schwarz

Lets' take the new **primes** program for a spin and do a simple comparison of its **speed** and **space** requirements with those of the **unfaithful sieve**. The new program is **slower** and uses **more space**.

```
> :{
| primes = [x | x <- [2..], divisors x == [x]]
| divisors x = [d | d <- [2..x], x `mod` d == 0]
| :}

> :set +s

> primes !! 10
31
(0.00 secs, 421,296 bytes)

> primes !! 100
547
(0.02 secs, 5,799,472 bytes)

> primes !! 1000
7927
(1.66 secs, 750,429,248 bytes)

> primes !! 10000
104743
(226.74 secs, 99,481,787,792 bytes)
```

```
> :{
| sieve (p : xs) = p : sieve [x | x <- xs, x `mod` p > 0]
| primes = sieve [2..]
| :}

> :set +s

> primes !! 10
31
(0.00 secs, 388,448 bytes)

> primes !! 100
547
(0.01 secs, 1,571,864 bytes)

> primes !! 1000
7927
(0.22 secs, 131,166,832 bytes)

> primes !! 10000
104743
(20.78 secs, 14,123,155,080 bytes)
```

Richard Bird's next **primes** program is a lot more interesting.

Before he can present it though, he has to explain (in this slide and the next) **how to construct an infinite list** of **composite numbers**.

It is possible to have an infinite list of infinite lists.

For example

```
multiples = [map (n*) [1..] | n <- [2..]]
```

defines an infinite list of infinite lists of numbers, the first three being

```
[2,4,6,8,…], [3,6,9,12,…], [4,8,12,16,…]
```

Suppose we ask whether the above list of lists can be merged back into a single list, namely `[2..]`.

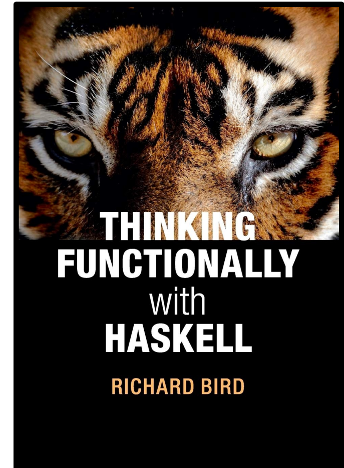
We can certainly merge two infinite lists:

```
merge :: Ord a => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x<y  = x:merge xs (y:ys)
                    | x==y = x:merge xs ys
                    | x>y  = y:merge (x:xs) ys
```

This version of merge removes duplicates. If the two arguments are in strictly increasing order, so is the result. Note the absence of any clauses of merge mentioning the empty list. Now it seems that if we define

```
mergeAll = foldr1 merge
```

then `mergeAll multiples` will return the infinite list `[2..]`. But it doesn't. What happens is that the computer gets stuck in an infinite loop trying attempting to compute the first element of the result…



Richard Bird

Now it seems that if we define

```
mergeAll = foldr1 merge
```

foldr1 is a variant on foldr restricted to nonempty lists.

then `mergeAll multiples` will return the infinite list `[2..]`. But it doesn't. What happens is that the computer gets stuck in an infinite loop trying attempting to compute the first element of the result, namely

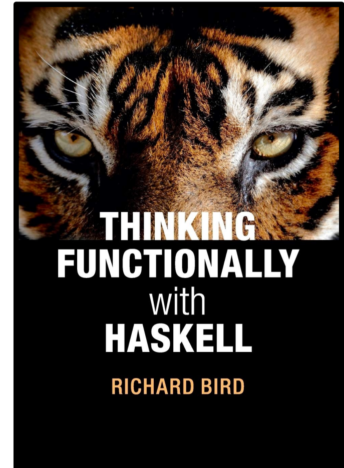
```
minimum (map head multiples)
```

It is simply **not possible** to compute the **minimum element** in an **infinite list**. Instead, we have to make us of the fact that `map head multiples` is in strictly increasing order, and define

```
mergeAll = foldr1 xmerge
xmerge (x:xs) ys = x:merge xs ys
```

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]      = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

With this definition, `mergeAll multiples` does indeed return.



Richard Bird

Let us now develop a **cyclic list** to generate an **infinite list** of all the **primes**.

To start with we define

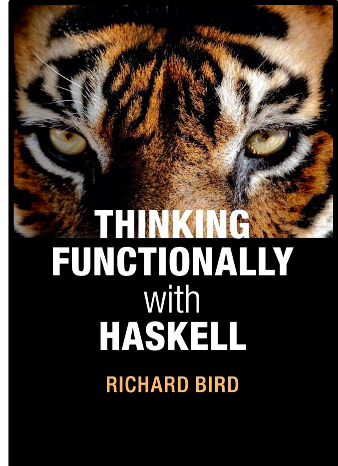
```
primes = [2..] \\ composites
composites = mergeAll multiples
multiples = [map (n*) [n..] | n <- [2..]]
```

where \\ subtracts one strictly increasing list from another

```
(x:xs) \\ (y:ys) | x<y  = x:(xs \\ (y:ys))
                 | x==y = xs \\ ys
                 | x>y  = (x:xs) \\ ys
```

Here, multiples consists of the list of all multiples of 2 from 4 onwards, all multiples of 3 from 9 onwards, all multiples of 4 from 16 onwards, and so on.

Merging the list, gives the infinite list of all the composite numbers, and taking its complement with respect to `[2..]` gives the primes.



Richard Bird

Richard Bird

So far so good, but the algorithm can be made many times faster by observing that too many multiples are being merged.

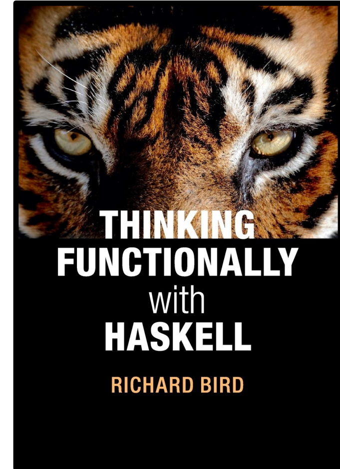
For instance, having constructed the multiples of 2 there is no need to construct the multiples of 4, or of 6, and so on.

What we really would like to do is just to construct the multiples of the primes.

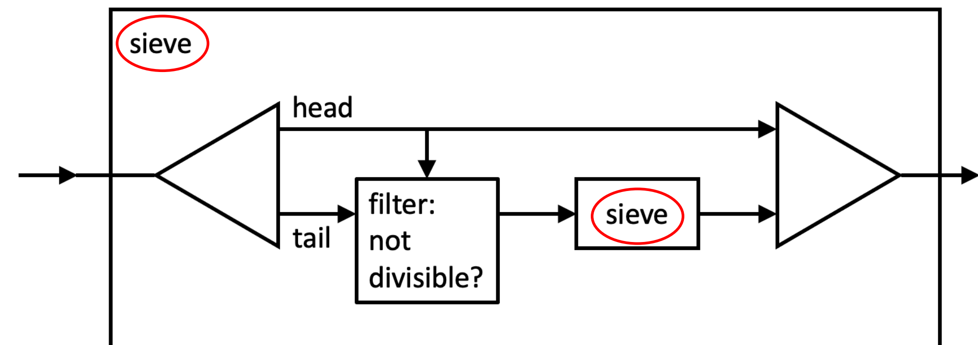
That leads to the idea of 'tying the recursive knot' and defining

```
primes = [2..] \\ composites
  where composites = mergeAll [map (p*) [p..] | p <- primes]
```

What we have here is a cyclic definition of primes.



The above notion of **tying the recursive knot** is reminiscent of the **cyclic** nature of the **stream** based **sieve** definition that we encountered in **Part 1**.



```
primes = [2..] \\ composites

  where composites = mergeAll [map (p*) [p..] | p <- primes]
```

It looks great, but does it work? Unfortunately, it doesn't: primes produces the undefined list.

In order to determine the first element of primes, the computation requires the first element of composites, which in turn requires the first element of primes.

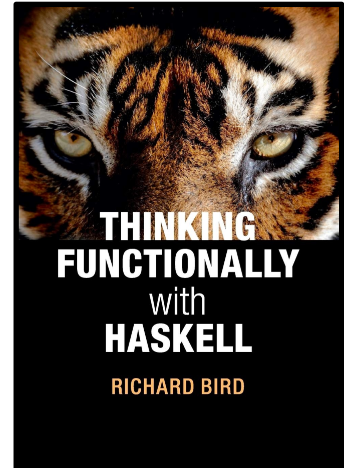
The computation gets stuck in an infinite loop.

To solve the problem we have to pump-prime (!) the computation by giving the computation the first prime explicitly.

We have to rewrite the definition as

```
primes = 2:([3..] \\ composites)

  where composites = mergeAll [map (p*) [p..] | p <- primes]
```

But this still doesn't produce the primes!



Richard Bird

The reason is a subtle one and is quite hard to spot. It has to do with the definition

```
mergeAll = foldr1 xmerge
```

The culprit is the function foldr1. Recall the Haskell definition:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

The order of the two defining equations is significant. In particular,

```
foldr1 f (x:undefined) = undefined
```

because the list argument is first matched against `x:[]`, causing the result to be undefined. That means

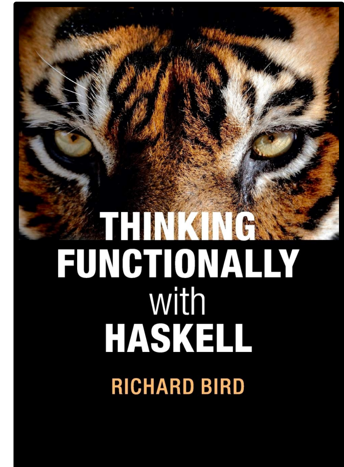
```
mergeAll [map (p*) [p..] | p <- 2:undefined] = undefined
```

What we wanted was

```
mergeAll [map (p*) [p..] | p <- 2:undefined] = 4:undefined
```

To effect this change we have to define `mergeAll` differently:

```
mergeAll (xs:xss) = xmerge xs (mergeAll xss)
```



Richard Bird

Richard Bird

Now we have:

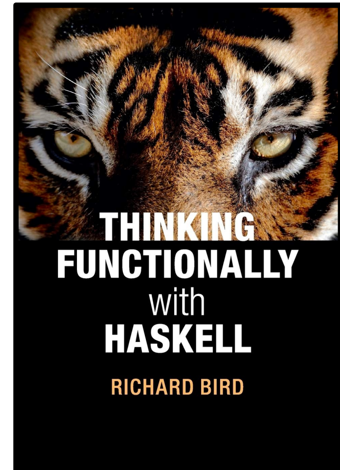
```
mergeAll [map (p*) [p..] | p <- 2:undefined]
= xmerge (map (2*) [2..]) undefined
= xmerge (4: map (2*) [3..]) undefined
= 4:merge (map (2*) [3..]) undefined
= 4:undefined
```

```
xmerge (x:xs) ys = x:merge xs ys
```

This version of mergeAll behaves differently on finite lists from the previous one.

With this final change we claim that primes does indeed get into gear and produce the primes.

…



On the next slide, as a recap, we see the whole program, and also do a simple comparison of its speed and space requirements with those of the first program.

```haskell
primes = (2:[3..] \\ composites)
  where composites = mergeAll [map (p*) [p..] | p <- primes]

(x:xs) \\ (y:ys) | x<y = x:(xs \\ (y:ys))
                 | x==y = xs \\ ys
                 | x>y  = (x:xs) \\ ys

mergeAll (xs:xss) = xmerge xs (mergeAll xss)

xmerge (x:xs) ys = x:merge xs ys

merge :: Ord a => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x<y  = x:merge xs (y:ys)
                    | x==y = x:merge xs ys
                    | x>y  = y:merge (x:xs) ys
```

```haskell
primes = [x | x <- [2..], divisors x == [x]]
divisors x = [d | d <- [2..x], x `mod` d == 0]
```

```
> primes !! 10
31
(0.03 secs, 383,624 bytes)

> primes !! 100
547
(0.00 secs, 737,784 bytes)

> primes !! 1000
7927
(0.03 secs, 8,701,248 bytes)

> primes !! 10000
104743
(0.56 secs, 193,131,088 bytes)

> primes !! 100000
1299721
(16.24 secs, 4,732,743,360 bytes)
```

```
> primes !! 10
31
(0.00 secs, 421,296 bytes)

> primes !! 100
547
(0.02 secs, 5,799,472 bytes)

> primes !! 1000
7927
(1.66 secs, 750,429,248 bytes)

> primes !! 10000
104743
(226.74 secs, 99,481,787,792 bytes)
```

```haskell
primes = (2:[3..] \\ composites)
  where composites = mergeAll [map (p*) [p..] | p <- primes]

(x:xs) \\ (y:ys) | x<y = x:(xs \\ (y:ys))
                 | x==y = xs \\ ys
                 | x>y  = (x:xs) \\ ys

mergeAll (xs:xss) = xmerge xs (mergeAll xss)

xmerge (x:xs) ys = x:merge xs ys

merge :: Ord a => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x<y  = x:merge xs (y:ys)
                    | x==y = x:merge xs ys
                    | x>y  = y:merge (x:xs) ys
```

Same as the previous slide, except that the smaller program is the **unfaithful sieve**.

```haskell
primes = sieve [2..]
sieve (p : xs) = p : sieve [x | x <- xs, x `mod` p > 0]
```

```
> primes !! 10
31
(0.03 secs, 383,624 bytes)

> primes !! 100
547
(0.00 secs, 737,784 bytes)

> primes !! 1000
7927
(0.03 secs, 8,701,248 bytes)

> primes !! 10000
104743
(0.56 secs, 193,131,088 bytes)

> primes !! 100000
1299721
(16.24 secs, 4,732,743,360 bytes)
```

```
> primes !! 10
31
(0.00 secs, 388,448 bytes)

> primes !! 100
547
(0.01 secs, 1,571,864 bytes)

> primes !! 1000
7927
(0.22 secs, 131,166,832 bytes)

> primes !! 10000
104743
(20.78 secs, 14,123,155,080 bytes)
```

What would Melissa O'Neill make of Richard Bird's primes program?

There is no need for us to speculate because the program is the subject of her paper's epilogue.

The Genuine Sieve of Eratosthenes

6 Epilogue

In discussing earlier drafts of this paper with other members of the functional programming community, I discovered that some functional programmers prefer to work solely with lists whenever possible, despite the ease with which languages such as Haskell and ML represent more advanced data structures.

Thus a frequent question from readers of earlier drafts whether a genuine Sieve of Eratosthenes could be implemented using only lists.

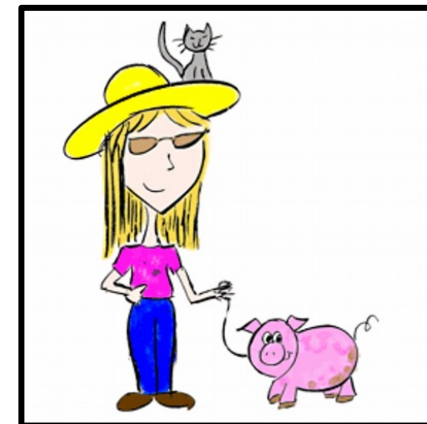
Some of those readers wrote their own implementations to show that you can indeed to so.

In a personal communication, Richard Bird suggested the following as a faithful list-based implementation of the Sieve of Eratosthenes.

This implementation maps well to the key ideas of this paper, so with his permission I have reproduced it.

The composites structure is our "table of iterators", but rather than using a tree or heap to represent the table, he uses a simple list of lists.

Each of the inner lazy lists corresponds to our "iterators". Removing elements from the front of the union of this list corresponds to removing elements from our priority queue.



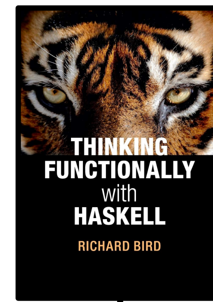
Melissa O'Neill

On the next slide we see the two programs by **Bird**, the one in the paper and the one in his book.

The programs clearly implement exactly the same algorithm, the only differences being some function names, the inlining of **multiples**, and the reduced accessibility / scope of two subordinate functions.

The Genuine Sieve of Eratosthenes

```haskell
primes = 2:([3..] 'minus' composites)
  where composites = union [multiples p | p <- primes]

multiples n = map (n*) [n..]

(x:xs) 'minus' (y:ys) | x<y  = x:(xs 'minus' (y:ys))
                      | x==y = xs 'minus' ys
                      | x>y  = (x:xs) 'minus' ys

union = foldr merge []
  where
    merge (x:xs) ys = x:merge' xs ys
    merge' (x:xs) (y:ys) | x<y  = x:merge' xs (y:ys)
                         | x==y = x:merge' xs ys
                         | x>y  = y:merge' (x:xs) ys
```




```haskell
primes = (2:[3..] \\ composites)
  where composites = mergeAll [map (p*) [p..] | p <- primes]

(x:xs) \\ (y:ys) | x<y  = x:(xs \\ (y:ys))
                 | x==y = xs \\ ys
                 | x>y  = (x:xs) \\ ys

mergeAll (xs:xss) = xmerge xs (mergeAll xss)

xmerge (x:xs) ys = x:merge xs ys

merge :: Ord a => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x<y  = x:merge xs (y:ys)
                    | x==y = x:merge xs ys
                    | x>y  = y:merge (x:xs) ys
```

Richard Bird

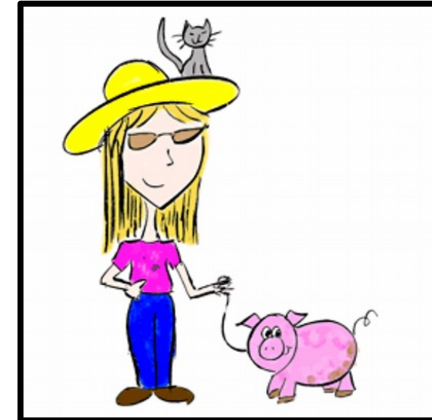
The Genuine Sieve of Eratosthenes

This code makes careful use of laziness. In particular, Bird remarks that “Taking the union of the infinite list of infinite lists [[4,6,8,10,..], [9,12,15,18..], [25,30,35,40,...],...] is tricky unless we exploit the fact that the first element of the result is the first element of the first infinite list. That is why union is defined in the way it is in order to be a productive function.”

While this incarnation of the Sieve of Eratosthenes does achieve the same ends as our earlier implementations, its list-based implementation does not give the same asymptotic performance. The structure of Bird’s table, in which the list of composites generated by the kth prime is the kth element in the outer list, means that when we are checking the ith number for primality, union requires $\pi(\sqrt{i})\ k=1\ k/pk \in \Theta(\sqrt{i}/(\log i)2)$ time, resulting in a time complexity of $\Theta(n \sqrt{n} \log \log n/(\log n)^2)$, making it asymptotically worse than trial division, but only by a factor of $\log \log n$.

In practice, Bird’s version is good enough for many purposes. His code is about four times faster than our trial-division implementation for small n, and because log log n grows very slowly, it is faster for all practical sizes of n. It is also faster than our initial tree-based code for n < 108.5, and faster than the basic priority-queue version for n < 275, 000, but never faster than the priority-queue version that uses the wheel. Incidentally, Bird’s algorithm could be modified to support the wheel optimizations, but the changes are nontrivial (in particular, multiples would need to take account of the wheel).

For any problem, there is a certain challenge in trying to solve it elegantly using only lists, but there are nevertheless good reasons to avoid too much of a fixation on lists, particularly if a focus on seeking elegant list-based solutions induces a myopia for elegant solutions that use other well-known data structures. For example, some of the people with whom I discussed the ideas in this paper were not aware that a solution using a heap was possible in a purely functional language because they had never seen one used in a functional context. The vast majority of well-understood standard data structures can be as available in a functional environment as they are in an imperative one, and in my opinion, we should not be afraid to be seen to use them.



Melissa O'Neill

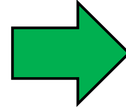
Algorithm	**Asymptotic Time Complexity**
Sieve of Eratosthenes	$\Theta(n \log \log n)$
Trial division	$\Theta(n \sqrt{n} / (\log n)^2)$
Unfaithful Sieve	$\Theta(n^2 / (\log n)^2)$
Richard Bird's Sieve	$\Theta(n \sqrt{n} \log \log n / (\log n)^2)$

Melissa O'Neill

Now let’s translate **Bird**’s program into **Scala**. (I have tweaked some function names a bit).

Again, because the **Haskell** version uses an **infinite list**, in **Scala** we use an **infinite lazy list**.

```haskell
primes = 2:([3..] 'minus' composites)

  where composites = union [multiples p | p <- primes]

multiples n = map (n*) [n..]

(x:xs) 'minus' (y:ys) | x<y  = x:(xs 'minus' (y:ys))
                      | x==y = xs 'minus' ys
                      | x>y  = (x:xs) 'minus' ys

union = foldr xmerge []

  where

    xmerge (x:xs) ys = x:merge xs ys

    merge (x:xs) (y:ys) | x<y  = x:merge xs (y:ys)
                        | x==y = x:merge xs ys
                        | x>y  = y:merge (x:xs) ys
```



```scala
def primes: LazyList[Int] =

  def composites = union { for p <- primes yield multiples(p) }

  2 #:: minus(LazyList.from(3), composites)

def multiples(n: Int) = LazyList.from(n) map (n * _)

val minus: (LazyList[Int], LazyList[Int]) => LazyList[Int] =
  case (x #:: xs, y #:: ys) =>
    if x<y then x #:: minus(xs,y#::ys)
    else if x==y then minus(xs,ys)
    else minus(x#::xs,ys)

def union(xss: LazyList[LazyList[Int]]): LazyList[Int] =

  def merge: (LazyList[Int], LazyList[Int]) => LazyList[Int] =
    case (x #:: xs, y #:: ys) =>
      if x<y then x #:: merge(xs,y#::ys)
      else if x==y then x #:: merge(xs,ys)
      else y #:: merge(x#::xs,ys)

  val xmerge: (LazyList[Int], LazyList[Int]) => LazyList[Int] =
    case (x #:: xs, ys) => x #:: merge(xs,ys)

  xss.foldRight(LazyList.empty[Int])(xmerge)
```

```scala
def primes: LazyList[Int] =

  def composites = union { for p <- primes yield multiples(p) }

  2 #:: minus(LazyList.from(3), composites)

def multiples(n: Int) = LazyList.from(n) map (n * _)

val minus: (LazyList[Int], LazyList[Int]) => LazyList[Int] =
  case (x #:: xs, y #:: ys) =>
    if x<y then x #:: minus(xs,y#::ys)
    else if x==y then minus(xs,ys)
    else minus(x#::xs,ys)

def union(xss: LazyList[LazyList[Int]]): LazyList[Int] =

  def merge: (LazyList[Int], LazyList[Int]) => LazyList[Int] =
    case (x #:: xs, y #:: ys) =>
      if x<y then x #:: merge(xs,y#::ys)
      else if x==y then x #:: merge(xs,ys)
      else y #:: merge(x#::xs,ys)

  val xmerge: (LazyList[Int], LazyList[Int]) => LazyList[Int] =
    case (x #:: xs, ys) => x #:: merge(xs,ys)

  xss.foldRight(LazyList.empty[Int])(xmerge)
```

Unfortunately the **Scala** program encounters a **StackOverflowError**.

As seen earlier, the **Haskell** program makes careful use of **laziness** to deal with problems like the following:

- "in order to determine the first element of **primes**, the computation requires the first element of **composites**, which in turn requires the first element of **primes**".
- "Taking the **union** of the **infinite list** of **infinite lists** [[4,6,8,10,..], [9,12,15,18..], [25,30,35,40,...],...] is **tricky** unless we exploit the fact that the first element of the result is the first element of the first infinite list. That is why **union** is defined in the way it is in order to be a **productive function**."

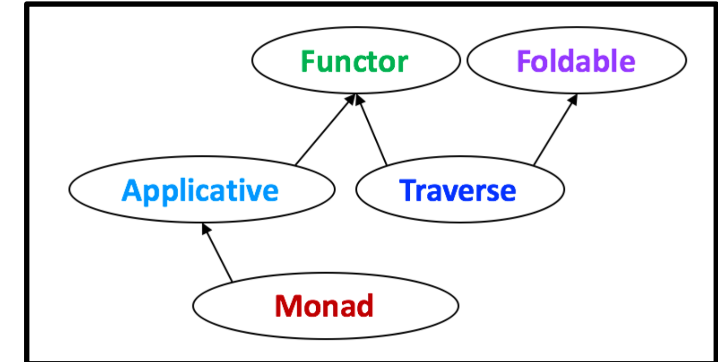
While the **Scala** program enlists the **laziness** of **LazyList**, it is defeated by the fact that while **Haskell**'s **right fold** over an **infinite list** can terminate if the **folded function** is **non-strict** in its right parameter, **Scala**'s **foldRight** function always **fails** to **terminate** when invoked on an **infinite list**.

primes calls **composites** which calls **union** which calls **foldRight**, but because the latter wants to **consume** all of the **infinite list** of **infinite lists** that it is passed, it calls **primes** again, which calls **composites** which calls **union** which calls **foldRight** again, and so on, these **nested calls** using more and more **stack space** until it runs out (see next slide for a section of the stack trace).

```
at scala.collection.immutable.LazyList$Deferrer$.$anonfun$$hash$colon$colon$extension$2(LazyList.scala:1142)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
at scala.collection.immutable.LazyList.$anonfun$mapImpl$1(LazyList.scala:516)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
at scala.collection.immutable.LazyList$LazyIterator.hasNext(LazyList.scala:1250)
at scala.collection.IterableOnceOps.reversed(IterableOnce.scala:1288)
at scala.collection.IterableOnceOps.reversed$(IterableOnce.scala:1285)
at scala.collection.AbstractIterable.reversed(Iterable.scala:926)
at scala.collection.IterableOnceOps.foldRight(IterableOnce.scala:665)
at scala.collection.IterableOnceOps.foldRight$(IterableOnce.scala:665)
at scala.collection.AbstractIterable.foldRight(Iterable.scala:926)
at UnfaithfulSieveFiniteList$package$.union(UnfaithfulSieveFiniteList.scala:86)
at UnfaithfulSieveFiniteList$package$.composites$1(UnfaithfulSieveFiniteList.scala:55)
at UnfaithfulSieveFiniteList$package$.primes$$anonfun$1(UnfaithfulSieveFiniteList.scala:56)
at scala.collection.immutable.LazyList$Deferrer$.$anonfun$$hash$colon$colon$extension$2(LazyList.scala:1142)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
at scala.collection.immutable.LazyList.$anonfun$mapImpl$1(LazyList.scala:516)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
at scala.collection.immutable.LazyList$LazyIterator.hasNext(LazyList.scala:1250)
at scala.collection.IterableOnceOps.reversed(IterableOnce.scala:1288)
at scala.collection.IterableOnceOps.reversed$(IterableOnce.scala:1285)
at scala.collection.AbstractIterable.reversed(Iterable.scala:926)
at scala.collection.IterableOnceOps.foldRight(IterableOnce.scala:665)
at scala.collection.IterableOnceOps.foldRight$(IterableOnce.scala:665)
at scala.collection.AbstractIterable.foldRight(Iterable.scala:926)
at UnfaithfulSieveFiniteList$package$.union(UnfaithfulSieveFiniteList.scala:86)
at UnfaithfulSieveFiniteList$package$.composites$1(UnfaithfulSieveFiniteList.scala:55)
at UnfaithfulSieveFiniteList$package$.primes$$anonfun$1(UnfaithfulSieveFiniteList.scala:56)
at scala.collection.immutable.LazyList$Deferrer$.$anonfun$$hash$colon$colon$extension$2(LazyList.scala:1142)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
at scala.collection.immutable.LazyList.$anonfun$mapImpl$1(LazyList.scala:516)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
at scala.collection.immutable.LazyList$LazyIterator.hasNext(LazyList.scala:1250)
at scala.collection.IterableOnceOps.reversed(IterableOnce.scala:1288)
at scala.collection.IterableOnceOps.reversed$(IterableOnce.scala:1285)
at scala.collection.AbstractIterable.reversed(Iterable.scala:926)
at scala.collection.IterableOnceOps.foldRight(IterableOnce.scala:665)
at scala.collection.IterableOnceOps.foldRight$(IterableOnce.scala:665)
at scala.collection.AbstractIterable.foldRight(Iterable.scala:926)
at UnfaithfulSieveFiniteList$package$.union(UnfaithfulSieveFiniteList.scala:86)
at UnfaithfulSieveFiniteList$package$.composites$1(UnfaithfulSieveFiniteList.scala:55)
at UnfaithfulSieveFiniteList$package$.primes$$anonfun$1(UnfaithfulSieveFiniteList.scala:56)
```

It turns out that switching from the **eager foldRight** function provided by the **Scala** standard library to the **lazy foldRight** function provided by **Cat**'s **Foldable** type class resolves the problem.

```
package cats
…
@typeclass trait Traverse[F[_]] extends Functor[F] with Foldable[F] … {
…

@typeclass trait Foldable[F[_]] … {
…
  /**
   * Left associative fold on 'F' using the function 'f'.
   *

  …
  */
  def foldLeft[A, B](fa: F[A], b: B)(f: (B, A) => B): B

  /**
   * Right associative lazy fold on `F` using the folding function 'f'.
   *
   * This method evaluates `lb` lazily (in some cases it will not be
   * needed), and returns a lazy value. We are using ` (A, Eval[B]) =>
   * Eval[B]` to support laziness in a stack-safe way. Chained
   * computation should be performed via .map and .flatMap.
   *
   * For more detailed information about how this method works see the
   * documentation for `Eval[_]`.
   …
   */
  def foldRight[A, B](fa: F[A], lb: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B]
```



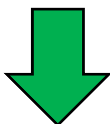
To use the **lazy right fold** we have to modify the **xmerge** function so that

- its second parameter, i.e. the **accumulator**, is an instance of the **Eval monad**
- its return type is also an instance of the **Eval monad**.

```
def union(xss: LazyList[LazyList[Int]]): LazyList[Int] =

  def merge: (LazyList[Int], LazyList[Int]) => LazyList[Int] = …

  val xmerge: (LazyList[Int], LazyList[Int]) => LazyList[Int] =
    case (x #:: xs, ys) => x #:: merge(xs,ys)

  xss.foldRight(LazyList.empty[Int])(xmerge)
```

```
def foldRight[B](z: B)(op: (A, B) => B): B
```

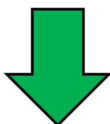
Scala standard library



```
def foldRight[A, B](fa: F[A], lb: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B]
```

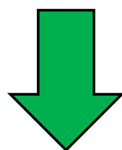
Cats' Foldable

```
val xmerge: (LazyList[Int], LazyList[Int]) => LazyList[Int]
```



```
val xmerge: (LazyList[Int], Eval[LazyList[Int]]) => Eval[LazyList[Int]]
```

```scala
def union(xss: LazyList[LazyList[Int]]): LazyList[Int] =

  def merge: (LazyList[Int], LazyList[Int]) => LazyList[Int] = …

  val xmerge: (LazyList[Int], LazyList[Int]) => LazyList[Int] =
    case (x #:: xs, ys) => x #:: merge(xs,ys)

  xss.foldRight(LazyList.empty[Int])(xmerge)
```

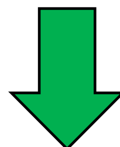


```scala
def union(xss: LazyList[LazyList[Int]]): LazyList[Int] =

  def merge: (LazyList[Int], LazyList[Int]) => LazyList[Int] = …

  import cats.{Foldable, Eval}

  val xmerge: (LazyList[Int], Eval[LazyList[Int]]) => Eval[LazyList[Int]] =
    case (x #:: xs, ysEval) => Eval.now(x #:: merge(xs,ysEval.value))

  Foldable[LazyList].foldRight(xss,Eval.now(LazyList.empty[Int]))(xmerge).value
```

Thanks to syntax extensions, rather than calling the foldRight function provided by the Foldable type class, we can call the foldr function provided by Foldable instances.

The latter function is called foldr so that it does not clash with the foldRight function defined in the Scala standard library.

```
def union(xss: LazyList[LazyList[Int]]): LazyList[Int] =

  def merge: (LazyList[Int], LazyList[Int]) => LazyList[Int] = …

  import cats.{Foldable, Eval}

  val xmerge: (LazyList[Int], Eval[LazyList[Int]]) => Eval[LazyList[Int]] =
    case (x #:: xs, ysEval) => Eval.now(x #:: merge(xs,ysEval.value))

  Foldable[LazyList].foldRight(xss,Eval.now(LazyList.empty[Int]))(xmerge).value
```



```
def union(xss: LazyList[LazyList[Int]]): LazyList[Int] =

  def merge: (LazyList[Int], LazyList[Int]) => LazyList[Int] = …

  import cats.Eval
  import cats.implicits._

  val xmerge: (LazyList[Int], Eval[LazyList[Int]]) => Eval[LazyList[Int]] =
    case (x #:: xs, ysEval) => Eval.now(x #:: merge(xs,ysEval.value))

  xss.foldr(Eval.now(LazyList.empty[Int]))(xmerge).value
```

```scala
def primes: LazyList[Int] =

  def composites = union { for p <- primes yield multiples(p) }

  2 #:: minus(LazyList.from(3), composites)

def multiples(n: Int) = LazyList.from(n) map (n * _)

val minus: (LazyList[Int], LazyList[Int]) => LazyList[Int] =
  case (x #:: xs, y #:: ys) =>
    if x<y then x #:: minus(xs,y#::ys)
    else if x==y then minus(xs,ys)
    else minus(x#::xs,ys)

def union(xss: LazyList[LazyList[Int]]): LazyList[Int] =

  def merge: (LazyList[Int], LazyList[Int]) => LazyList[Int] = …

  val xmerge: (LazyList[Int], Eval[LazyList[Int]]) => Eval[LazyList[Int]] =
    case (x #:: xs, ysEval) => Eval.now(x #:: merge(xs,ysEval.value))

  xss.foldr(Eval.now(LazyList.empty[Int]))(xmerge).value
```

```scala
import cats.Eval
import cats.implicits._
```

```scala
List(1_000, 10_000, 50_000, 100_000).foreach {
  n => println(s"$n => ${eval(primes(n))}")
}
```

```
1000 => (7927,35 milliseconds)
10000 => (104743,318 milliseconds)
50000 => (611957,3160 milliseconds)
100000 => (1299721,7464 milliseconds)
```

```scala
println(primes.take(100).toList)
```

```
List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109,
113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241,
251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389,
397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541)
```

That’s it for **Part 2**.

I hope you found that useful.