# Getting Nice Examples
# (with less greedy algorithms)

https://bit.ly/hypothesis-simplification-bigo

# Who am I?

- David R. MacIver. The R is for namespacing purposes.
- I wrote a testing library for Python called Hypothesis.
- It was harder than I expected...

# The Problem

- Property based testing.
- Throw random structured data at your tests, see what breaks.
- ...but random data is messy and hard to read.
- So we have to shrink it down. How?

# Starting point: Quickcheck

- A shrink function iterates over simpler versions of an example.
- Greedily apply it until no simpler example breaks the test.

Here's some Python:

```
def greedy_shrink(ls, constraint, shrink):
    while True:
        for s in shrink(ls):
            if constraint(s):
                ls = s
                break
        else:
            return ls
```

# What's wrong with this?

- Not much. It's worked pretty well for Quickcheck for 15+ years.
- But Hypothesis is different...
  - Tests are slower (Python, more complex tests)
  - Examples are larger.
- Some pathologically bad cases for this algorithm.
- So we want to improve worst case behaviour.

# So what to do about it?

- We'll start by analyzing some concrete examples.
- We'll focus on lists of non-negative integers.
- We'll minimize just subject to some abstract constraint.
- "Minimizing" here means try to get shorter lists of smaller values.
- We want to reduce the number of times we call the predicate.
- We'll start by looking at what makes a good shrink function.
- Then we'll see why the greedy algorithm isn't *quite* the right one.

```
def shrink1(ls):
    for i in range(len(ls)):
        s = list(ls)
        if s[i] > 0:
            s[i] -= 1
            yield list(s)
        del s[i]
        yield list(s)
```

# `len(ls) >= 2`

```
[5, 5]
[4, 5]
[3, 5]
[2, 5]
[1, 5]
[0, 5]
[0, 4]
[0, 3]
[0, 2]
[0, 1]
[0, 0]
```

9 shrinks with 17 function calls

What if we'd started from [100, 100]?

# Guiding Principle

Start from the simple end.

```
def shrink2(ls):
    for i in range(len(ls)):
        s = list(ls)
        del s[i]
        yield list(s)
        for x in range(ls[i]):
            s = list(ls)
            s[i] = x
            yield s
```

# `len(ls) >= 2`

```
[5, 5]
[0, 5]
[0, 0]
```

2 shrinks with 7 function calls

✓

```
sum(ls) >= 500
```

```
[1000]
[500]
```

1 shrink with 1003 function calls

# Guiding Principle

Don't generate too many shrinks.

(Instead let subsequent shrinks narrow it down.)

```
def shrink_integer(n):
    if not n:
        return
    for k in range(64):
        probe = 2 ** k
        if probe >= n:
            break
        yield probe - 1
    probe //= 2
    while True:
        probe = (probe + n) // 2
        yield probe
        if probe == n - 1:
            break

def shrink3(ls):
    for i in range(len(ls)):
        s = list(ls)
        del s[i]
        yield list(s)
        for x in shrink_integer(ls[i]):
            s = list(ls)
            s[i] = x
            yield s
```

```
sum(ls) >= 500
```

```
[1000]
[511]
[503]
[501]
[500]
```

✓

4 shrinks, 79 function calls.

# So how are we doing?

- Pretty good at small lists with large numbers.
- But what about large lists?

# sum(ls) >= 3

```
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
(...)
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2]
[2, 2, 2, 2]
[2, 2, 2]
[2, 2]
[1, 2]
```

19 shrinks with 26 function calls.

```
def shrink4(ls):
    i = 1
    while i < len(ls):
        yield ls[:i]
        yield ls[len(ls) - i:]

    for i in range(len(ls)):
        s = list(ls)
        del s[i]
        yield list(s)
        for x in shrink_integer(ls[i]):
            s = list(ls)
            s[i] = x
            yield s
```

```
def shrink_to_prefix(ls):
    i = 1
    while i < len(ls):
        yield ls[:i]
        i *= 2

def shrink_individual_elements(ls):
    for i in range(len(ls)):
        s = list(ls)
        del s[i]
        yield list(s)
        for x in shrink_integer(ls[i]):
            s = list(ls)
            s[i] = x
            yield s

def shrink4(ls):
    yield from shrink_to_prefix(ls)
    yield from shrink_individual_elements(ls)
```

# `sum(ls) >= 3`

```
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[2, 2]
[1, 2]
```

2 shrinks with 12 function calls

✓

```
len([t for t in ls if t >= 5]) >= 10
```

```
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[7, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[5, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[5, 7, 20, 20, 20, 20, 20, 20, 20, 20]
[5, 5, 20, 20, 20, 20, 20, 20, 20, 20]
(...)
[5, 5, 5, 5, 5, 5, 5, 5, 5, 20]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 7]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```

22 shrinks with 695 function calls

```
def shrink_shared(ls):
    shared_indices = {}
    for i in range(len(ls)):
        shared_indices.setdefault(ls[i], []).append(i)
    for sharing in shared_indices.values():
        if len(sharing) > 1:
            for v in shrink_integer(ls[sharing[0]]):
                s = list(ls)
                for i in sharing:
                    s[i] = v
                yield s

def shrink5(ls):
    yield from shrink_to_prefix(ls)
    yield from shrink_shared(ls)
    yield from shrink_individual_elements(ls)
```

```
len([t for t in ls if t >= 5]) >= 10
```

```
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```

4 shrinks with 93 function calls

✓?

```
len([t for t in ls if t >= 5]) >= 10
```

```
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
[21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
[22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
[7, 23, 24, 25, 26, 27, 28, 29, 30, 31]
(...)
[5, 5, 5, 5, 5, 5, 5, 5, 7, 31]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 31]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 7]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```

22 shrinks with 763 function calls

```
def replace_with_simpler(ls):
    if not ls:
        return
    values = set(ls)
    values.remove(max(ls))
    values = sorted(values)
    for v in values:
        yield [min(v, l) for l in ls]

def shrink6(ls):
    yield from shrink_to_prefix(ls)
    yield from replace_with_simpler(ls)
    yield from shrink_shared(ls)
    yield from shrink_individual_elements(ls)
```

```
len([t for t in x if t >= 5]) >= 10
```

```
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```

5 shrinks with 98 function calls

✓

# Guiding Principle

Earlier shrinks should produce good input to later shrinks.

# Statistical intermission

- We've seen a lot of hand-crafted examples, but does this really work in the general case? Let's find out!
- We'll investigate behaviour of shrinkers 2 through 6.
- Generate lists of uniformly distributed unsigned 32-bit integers, with lengths drawn from [0, 100] uniformly at random.
- For each condition, draw lists until you get 1000 satisfying that condition.
- Using a fixed seed, so for a given condition each simplifier is run on the same data set.
- Now run each simplifier according to that condition measure the maximum number of function calls on that sample.
- This is not proper statistics! Sorry. I was in a hurry.

# Statistical intermission

| **Condition** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** |
|---|---|---|---|---|---|
| length >= 2 | 105 | 105 | 12 | 12 | 9 |
| sum >= 500 | 1102 | 178 | 80 | 80 | 80 |
| sum >= 3 | 108 | 107 | 9 | 9 | 9 |
| At least 10 >= 5 | 490 | 690 | 719 | 787 | 138 |

So 3 through 5 are ambiguous, but 6 is clearly the best.

# `len(set(ls)) >= 10`

```
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
[0, 101, 102, 103, 104, 105, 106, 107, 108, 109]
[0, 1, 102, 103, 104, 105, 106, 107, 108, 109]
[0, 1, 3, 103, 104, 105, 106, 107, 108, 109]
[0, 1, 2, 103, 104, 105, 106, 107, 108, 109]
(...)
[0, 1, 2, 3, 4, 5, 6, 7, 11, 109]
[0, 1, 2, 3, 4, 5, 6, 7, 9, 109]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 109]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 15]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 11]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

20 shrinks with 776 function calls

```
def greedy_shrink_with_dedupe(ls, constraint, shrink, seen=None):
    if seen is None:
        seen = set()
    while True:
        for s in shrink(ls):
            key = tuple(s)
            if key in seen:
                continue
            seen.add(key)
            if constraint(s):
                ls = s
                break
        else:
            return ls
```

Doesn't change behaviour, but down to 528 function calls.

# Statistical intermission

| Condition | Normal | Deduped |
|---|---|---|
| length >= 2 | 9 | 6 |
| sum >= 500 | 80 | 35 |
| sum >= 3 | 9 | 6 |
| At least 10 >= 5 | 138 | 95 |
| At least 10 distinct | 1065 | 777 |

# So why is this so bad?

- We're performing a lot of useless operations.
- We keep retrying useless simplifications on each change.
- Lets see if we can stop doing that...

```
def multicourse_shrink1(ls, constraint, seen):
    if seen is None:
        seen = set()
    for shrink in [
        shrink_to_prefix,
        replace_with_simpler,
        shrink_shared,
        shrink_individual_elements,
    ]:
        ls = greedy_shrink_with_dedupe(
            ls, constraint, shrink, seen)
    return ls
```

# `len(set(ls)) >= 10`

```
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
[0, 101, 102, 103, 104, 105, 106, 107, 108, 109]
[0, 1, 102, 103, 104, 105, 106, 107, 108, 109]
[0, 1, 3, 103, 104, 105, 106, 107, 108, 109]
[0, 1, 2, 103, 104, 105, 106, 107, 108, 109]
[0, 1, 2, 3, 104, 105, 106, 107, 108, 109]
[0, 1, 2, 3, 7, 105, 106, 107, 108, 109]
(...)
[0, 1, 2, 3, 4, 5, 6, 107, 108, 109]
[0, 1, 2, 3, 4, 5, 6, 7, 108, 109]
[0, 1, 2, 3, 4, 5, 6, 7, 15, 109]
[0, 1, 2, 3, 4, 5, 6, 7, 11, 109]
[0, 1, 2, 3, 4, 5, 6, 7, 9, 109]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 109]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 15]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 11]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

20 shrinks with 436 function calls

```
def shrink_index(i):
    def accept(ls):
        if i >= len(ls):
            return
        for v in shrink_integer(ls[i]):
            s = list(ls)
            s[i] = v
            yield s
    return accept

def delete_single_elements(ls):
    for i in range(len(ls)):
        s = list(ls)
        del s[i]
        yield s

def shrinkers_for(ls):
    yield shrink_to_prefix
    yield delete_single_elements
    yield replace_with_simpler
    yield shrink_shared
    for i in range(len(ls)):
        yield shrink_index(i)
```

```
def multicourse_shrink2(ls, constraint, seen=None):
    if seen is None:
        seen = set()
    for shrink in shrinkers_for(ls):
        ls = greedy_shrink_with_dedupe(
            ls, constraint, shrink, seen)
    return ls
```

Same shrink pattern, but only 75 calls now. Yay!

But unfortunately it's wrong.

```
len(ls) >= 2 and x[0] > x[1]
```

```
[101, 100]
[101, 0]
```

1 shrink with 16 function calls

# What happened?

Shrinking the second element enabled further shrinking the first, but in this implementation we never go back to previous indices...

```
def multicourse_shrink3(ls, constraint, seen=None):
    if seen is None:
        seen = set()
    while True:
        new_ls = ls
        for shrink in shrinkers_for(ls):
            new_ls = greedy_shrink_with_dedupe(
                new_ls, constraint, shrink, seen)
        if ls == new_ls:
            return ls
        ls = new_ls
```

```
[101, 100]
[101, 0]
[1, 0]
```

2 shrinks with 20 function calls

Rerunning the previous example now gives us 131 function calls. It would be nice to have it faster yet, but that's probably about as much as we can reasonably expect.

# Statistical intermission

| Condition | Single-pass | Multi-pass |
|---|---|---|
| length >= 2 | 6 | 6 |
| sum >= 500 | 35 | 35 |
| sum >= 3 | 6 | 6 |
| At least 10 >= 5 | 95 | 73 |
| At least 10 distinct | 777 | 131 |
| First > Second | 1506 | 1470 |

# So how are we doing?

- Pretty good!
- We've gotten very good at exploiting structure.
- This lets us handle large lists by rapidly finding smaller ones.
- But what happens when there is no structure?

# A messy condition

```
import hashlib

def messy_condition(ls):
    return hashlib.md5(
      repr(ls).encode('utf-8')).hexdigest()[0] == '0'
```

Worst case is multi-pass takes 9007 vs 958 for single pass greedy algorithm.

The problem is that the condition is very sensitive to changes, so later changes basically *always* unlock possible earlier changes. Greedy algorithm can exploit this. Multi-pass cannot.

But honestly who cares?

# What's new here?

- Somewhere between "none of it" and "most of it".
- Unusual to see this sort of work in a Quickcheck.
- Theft (Quickcheck for C) also does deduplication.
- Multipass shrinking was inspired by afl-tmin.
- Some of the principles of good shrinking come straight from Quickheck itself (or would have if I'd been paying attention).
- Some of the heuristics were inspired by the implementation of timsort.
- The cloning heuristics are either novel or an independent reinvention.

# Why is this unusual?

- Because it wasn't needed.
- And because I omitted all the other hard parts.
- When working with lists of arbitrary data rather than integers, you can't do many of these.
- ...which is why Hypothesis works on an intermediate representation.

# Testing

You should be using this sort of testing. It will save you effort *and* improve the quality of your software.

- Hypothesis (Python)
- jsverify (JavaScript)
- theft (C)
- ScalaCheck (Scala)
- test.check (Clojure)
- FsCheck (F#)
- QuickCheck (Haskell)
- QuickCheck (Erlang)

# Obligatory plug

- I offer Hypothesis training and support (but you should start without it).
- I'm also potentially available for other contracting work.

drmaciver.com / @DRMacIver

https://hypothesis.readthedocs.org/

https://bit.ly/hypothesis-simplification-bigo