# Cute Algorithms!

稲葉 一浩

at #spro12

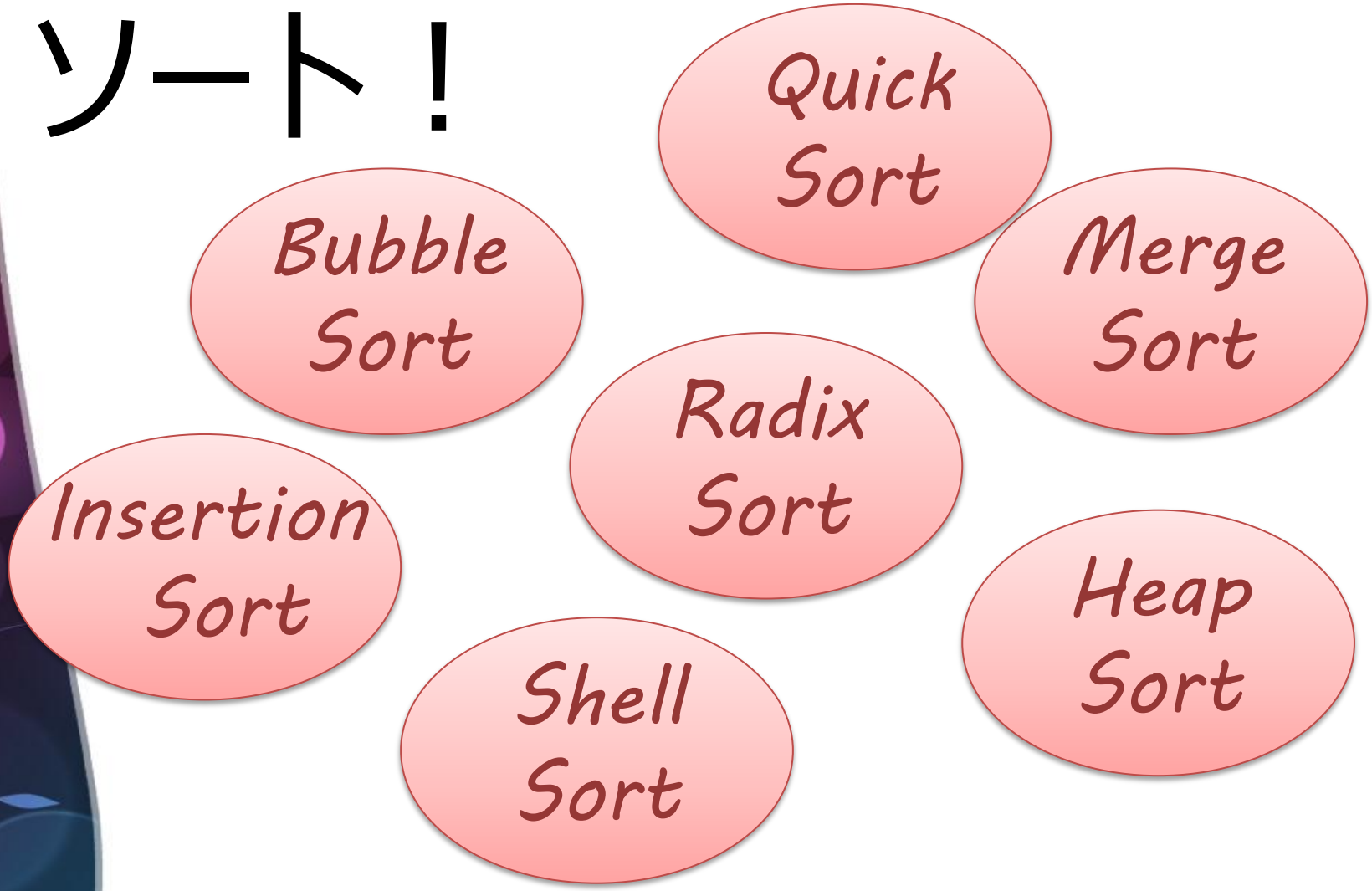
まずは準備運動です！

# SLOWEST SORT

Thanks to: @y_benjo @rng_58 @hos_lyric @oxy @xhl_kogitsune @args1234

# アルゴリズムと言えば！

# ソート！



# ソートと言えば！

O(n log n)！

Quick Sort

Ins Sort

Shell Sort

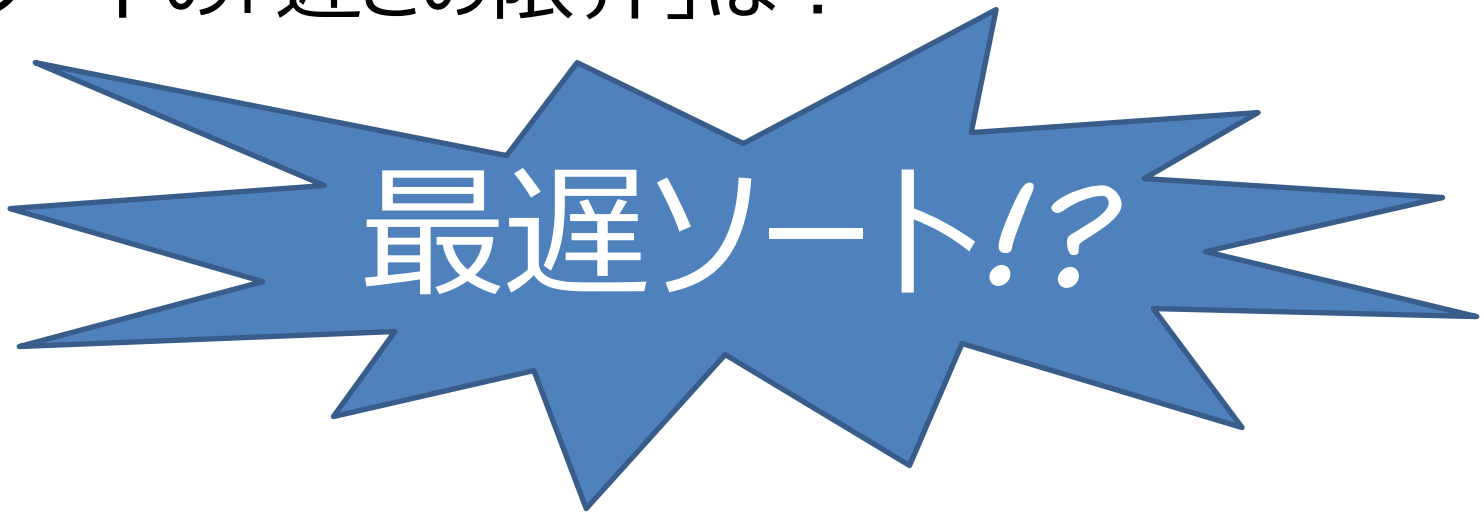
Heap Sort

## 比較ソートの理論限界 [編集]

計算理論において、n個のデータのソートは、データの大小比較のみによって行う場合、最悪計算量が最低でも O(n log n) 必要なことが知られている。

# では、逆に・・・

ソートの「遅さの限界」は？




一番たくさん大小比較を実行するソート法は？

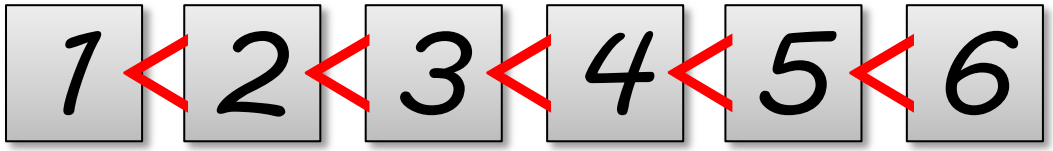
# 「最も遅い」ソート法は？

- よくある答え：　**ボゴソート**
  - ランダムに並替→ソートされたら終了　のループ
- 問題点：　**ずるい。**
  - 無駄に同じ比較を繰り返しているだけ。
  - それがありなら幾らでも人工的に遅くできる。

「無駄」な比較はノーカウントで、
一番多く大小比較をするソートアルゴリズムは？

# Bubble Sort

「最悪」$O(n)$ ： 入力がソート済みのとき

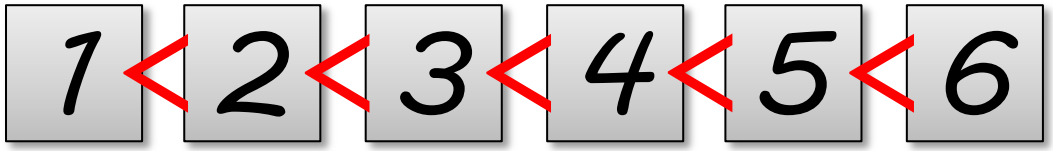



**速すぎる！**

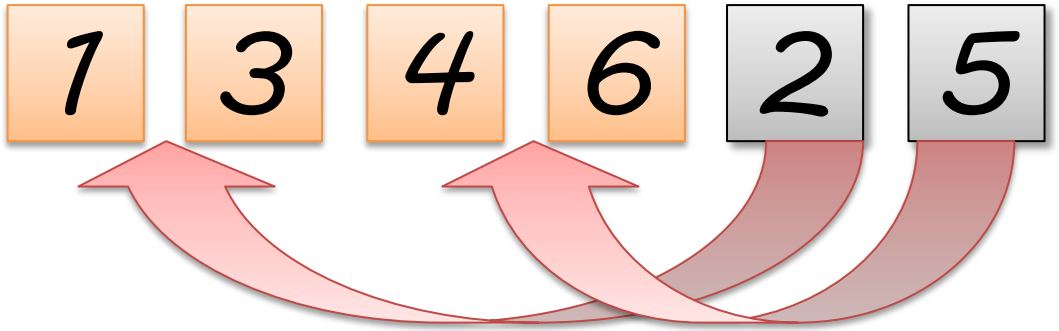
```
function bubble_sort(A) {
 for(i ← 0..n-2)
  for(j ← 1..n-1)
   if(A[j-1] > A[j])
    swap(A[j-1], A[j])
}
```

# Insertion Sort

「最悪」$O(n)$



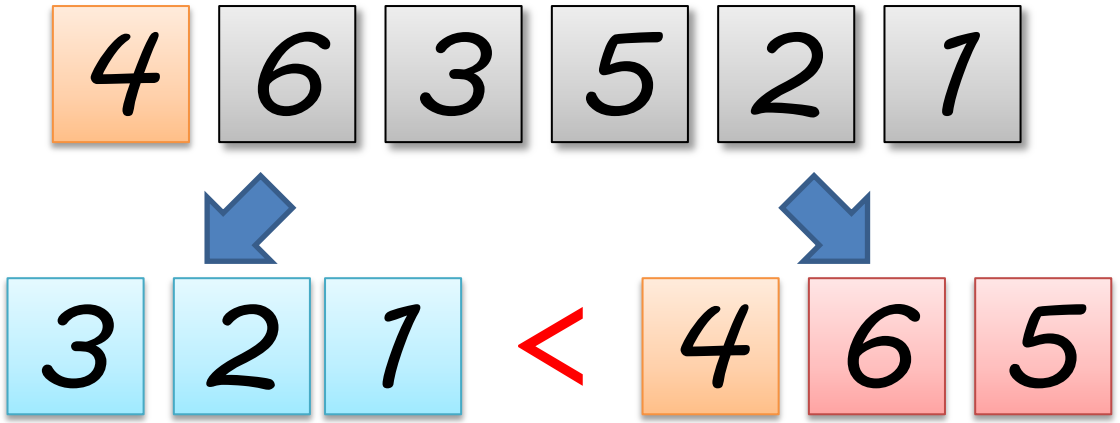

「平均」は $\Omega(n^2)$　挿入時平均で真中まで動く



# Quick Sort

「最悪」$\Omega(n \log n)$ **安定して遅い！**

– 証明: *Quick Sort* は無駄比較をしない。



# 理論限界は？

| | Bubble | Insert | Quick | Merge | 理論限界 |
|---|---|---|---|---|---|
| 最悪 | $n$ | $n$ | $n \log n$ | $n \log n$ | $n \log n$ |
| 平均 | ??? | $n^2$ | $n \log n$ | $n \log n$ | $n^2$ |

- どんなに遅くても「平均」は $O(n^2)$
  - $n(n-1)/2$ 通りしか比較するものがない
- どんなに遅くても「最悪」は $O(n \log n)$
  - 「最速」の$O(n \log n)$とほぼ完全に同じ証明

# 最遅ソート：理論限界を達成

*Quick Sort + Insertion Sort !*

```
function slow_sort(A) {
  // 最悪 O(n log n)
  quick_sort(A[0 ... n/2])
  // 平均 O(n²)
  insertion_sort(A[0...n])
}
```

# ここがCute!

- 「逆」の問題でも同じテクニックを応用
  - 「最速」計算量の下限と「最遅」の上限が同じように証明できる。
  - 速度を求めて実用上よく使われる組み合わせ
    Quick Sort + Insertion Sort
    が「最遅ソート」にも使える。

# 最遅ソートの応用

- N チームの総当たり戦を行います。
- 強弱にはだいたい順序がつくと仮定します。

「消化試合」（順位に影響しないことが確定している試合）をできるだけ減らす試合順は？

| | Tim | Shell | Comb | Stooge |
|---|---|---|---|---|
| Tim | - | ○ | | |
| Shell | × | - | | × |
| Comb | | | - | |
| Stooge | | ○ | | - |

非破壊型キュー

# FUNCTIONAL QUEUE

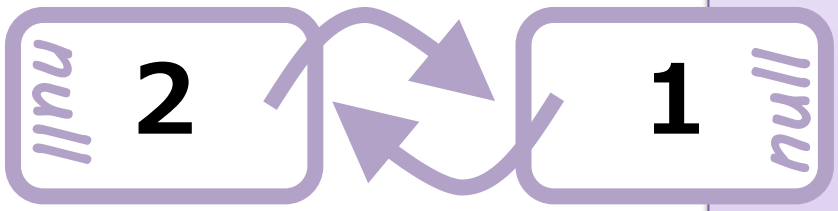
出典: “Purely Functional Data Structures”, Chris Okasaki

# Queue


- *push(x):* キューに値 *x* を入れる
- *pop()* ：キューから値を *1* 個取り出す
- 入れたのと同じ順番で出てくる

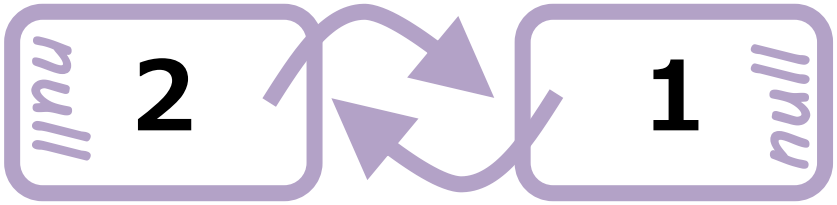
# よくある実装

*Linked List* で表現

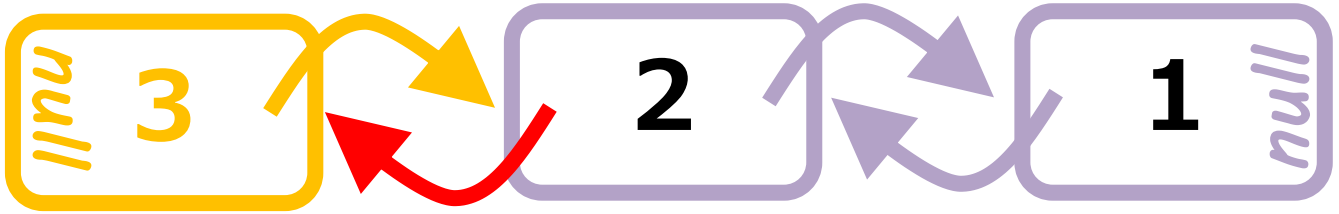



```
class Queue<T> {
  class Node {
    Node prev;
    T    value;
    Node next;
  }
  Node head;
  Node tail;
}
```

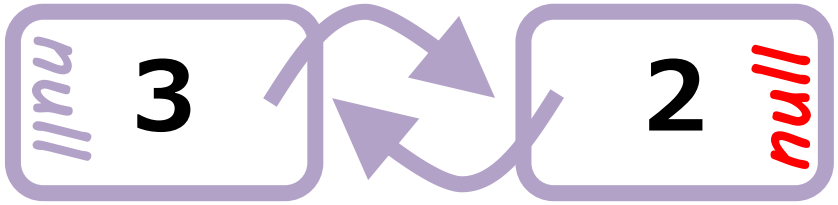
# Linked List Queue




## push 3




## pop




1

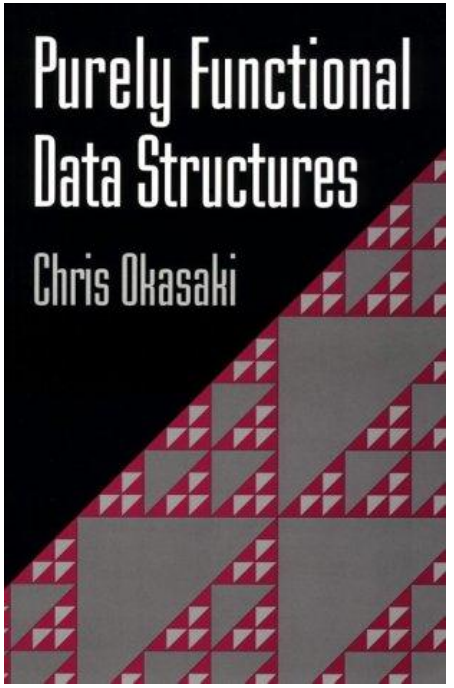
今日ご紹介する

# “Functional Queue”

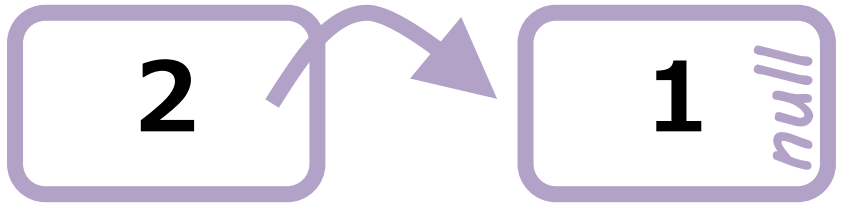
出典: “Purely Functional Data Structures”

**「一度作ったデータは絶対に書き換えない」**
データ構造に関するバイブル

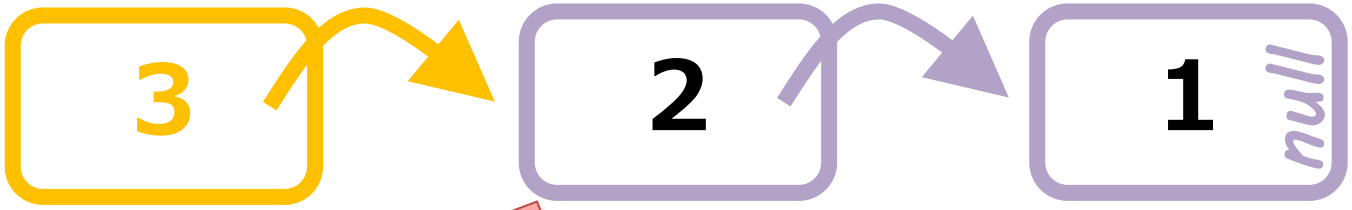
```
class Queue<T> {
  class Node {
    const T    value;
    const Node next;
  }
  const Node head;
  const Node tail;
  Queue push(T v);
}
```



# (1) Singly Linked List を使う




push 3

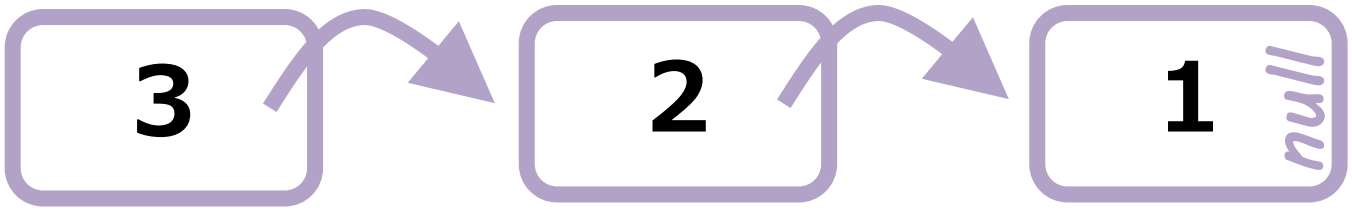



SingleLinkなら
先頭への追加は
書き換え不要

push前の状態が
残ってるので
参照できる

```
class Node {
  T     value;
  Node next;
}
```

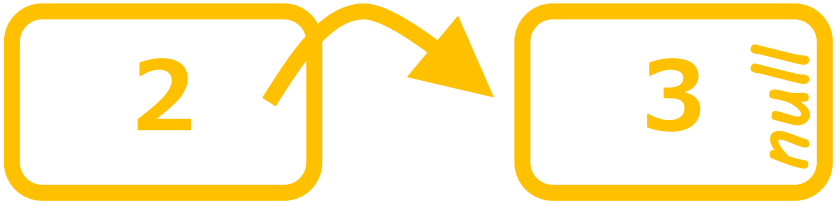
# (2) pop？



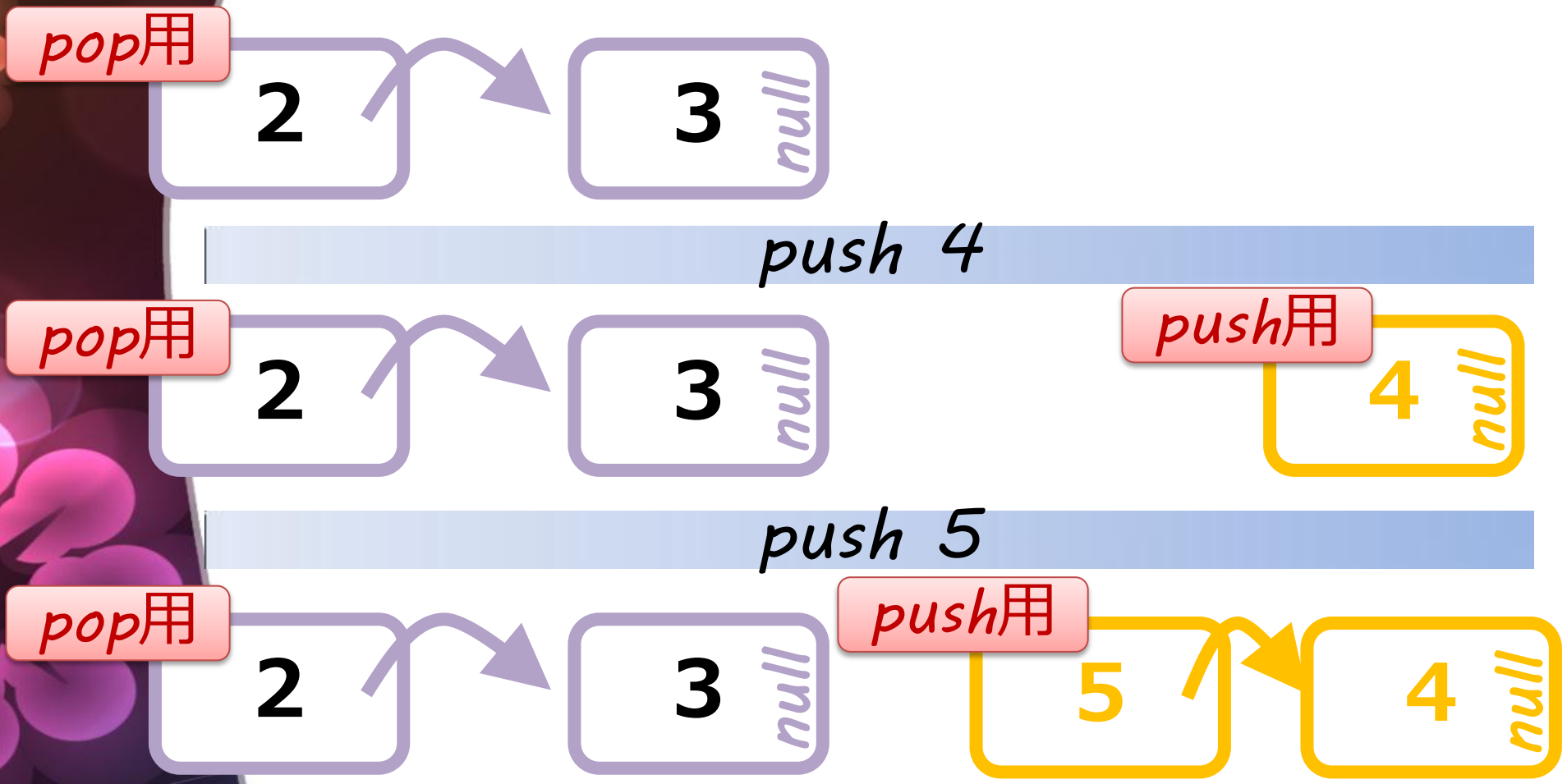

pop


Reverseしたリストを新しく作ってから、先頭Nodeを忘れる

1



# (3) もう一回 push？→ リストを２つ持つ

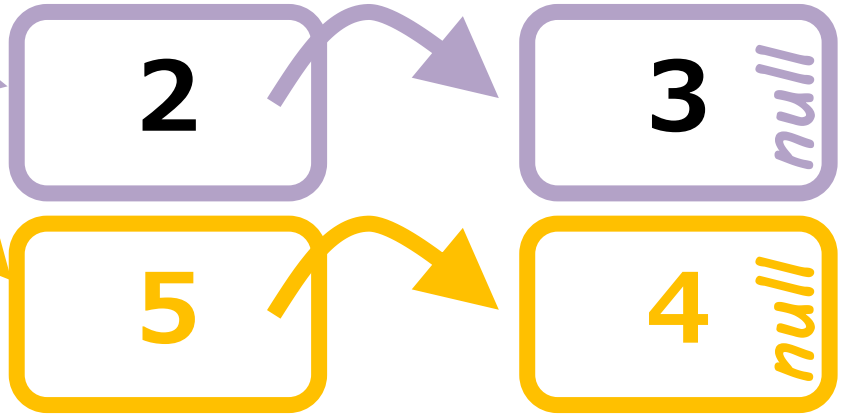


# ここまでの実装

*Singly Linked List* ２個で実装します

```
class Queue<T> {
  class Node {
    T    value;
    Node next;
  }
  Node for_pop;
  Node for_push;
}
```

*for_pop が空の時*
*pop されたら*
*for_push を reverse*



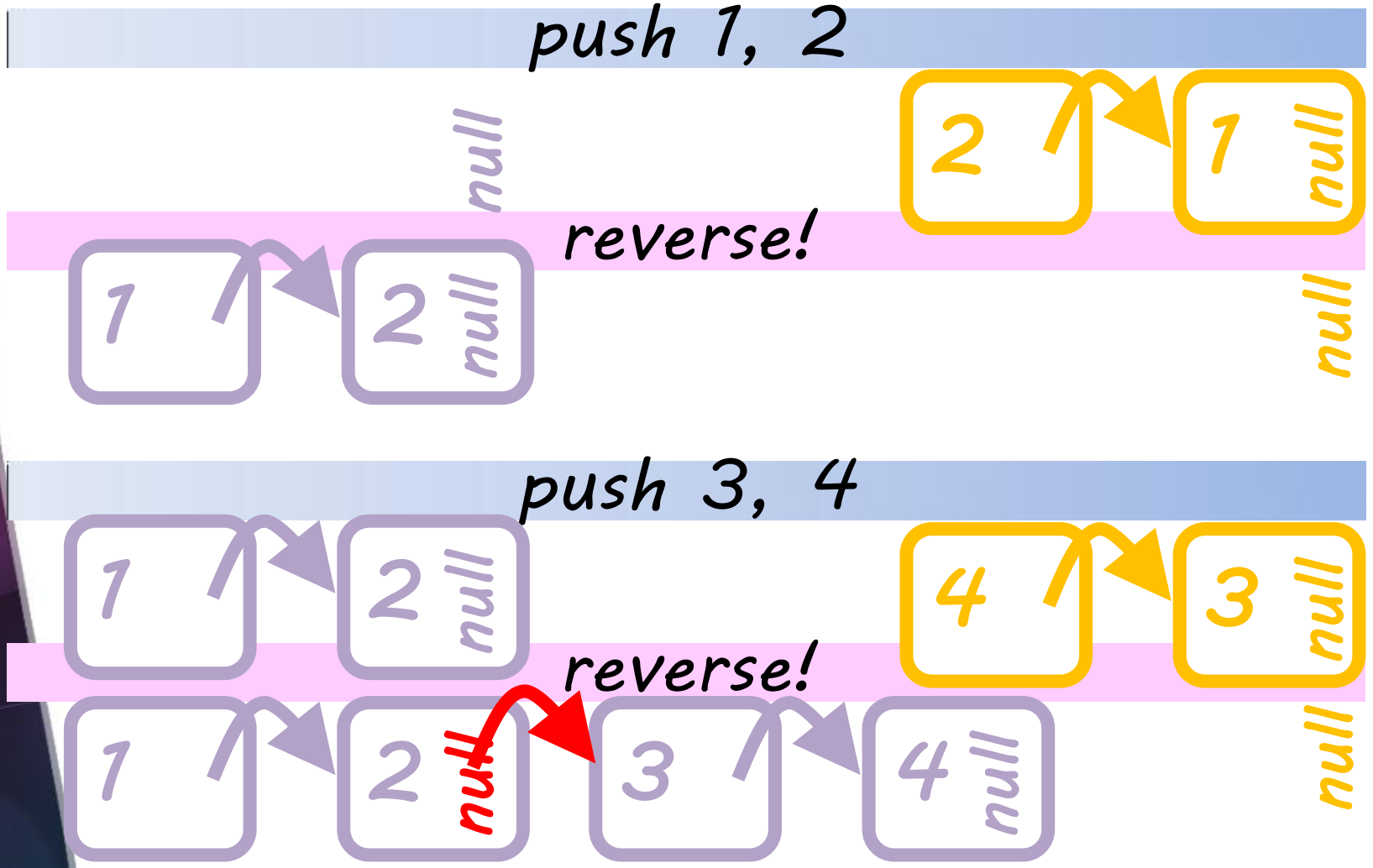
# 問題点

- for_pop が空の時だけ急にpopが遅い！

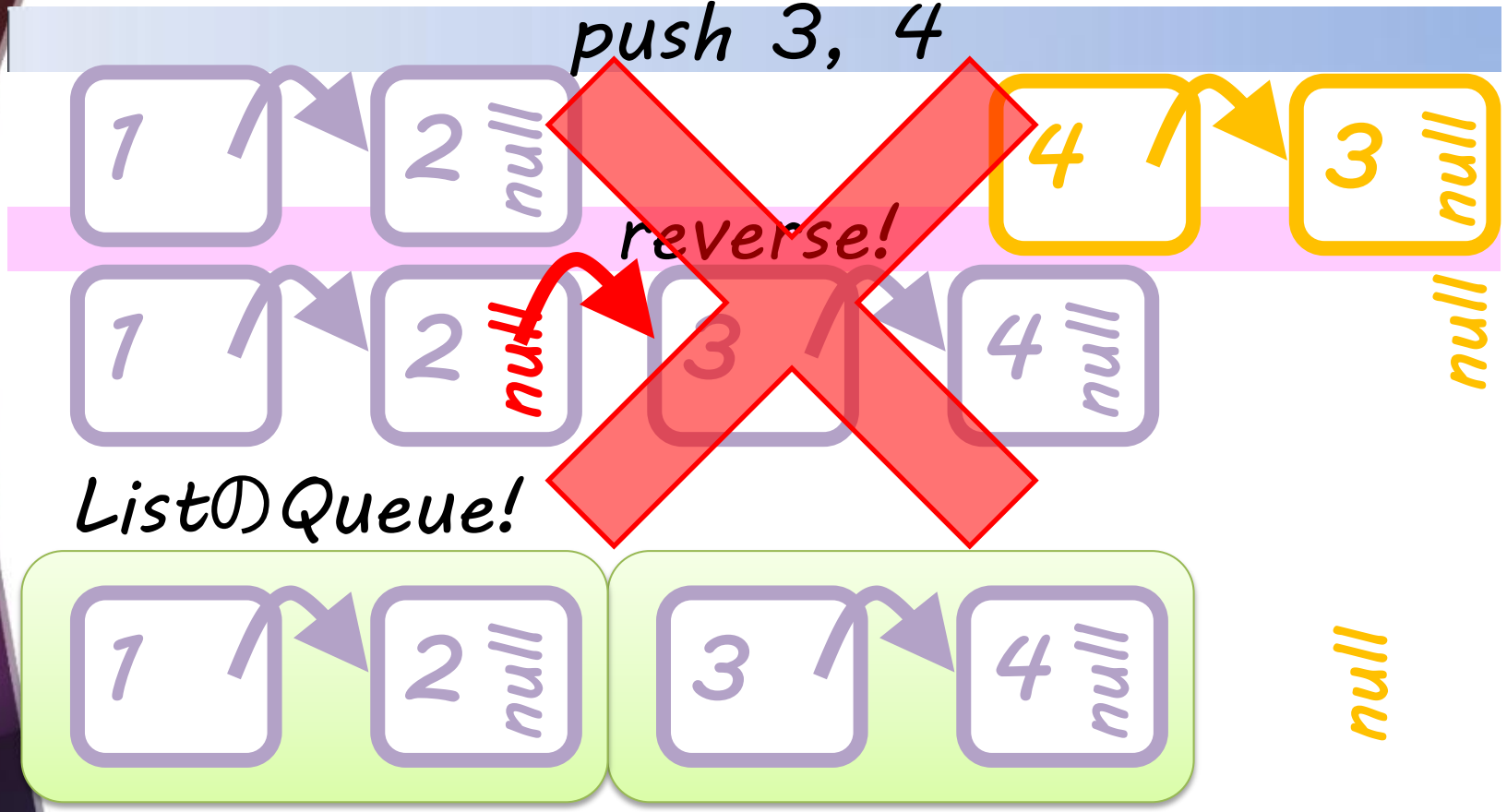
for_pop が空の時
pop されたら
for_push を reverse

# (4) 対策：小分けにreverse

for_pop が空じゃなくても、こまめにreverse ！

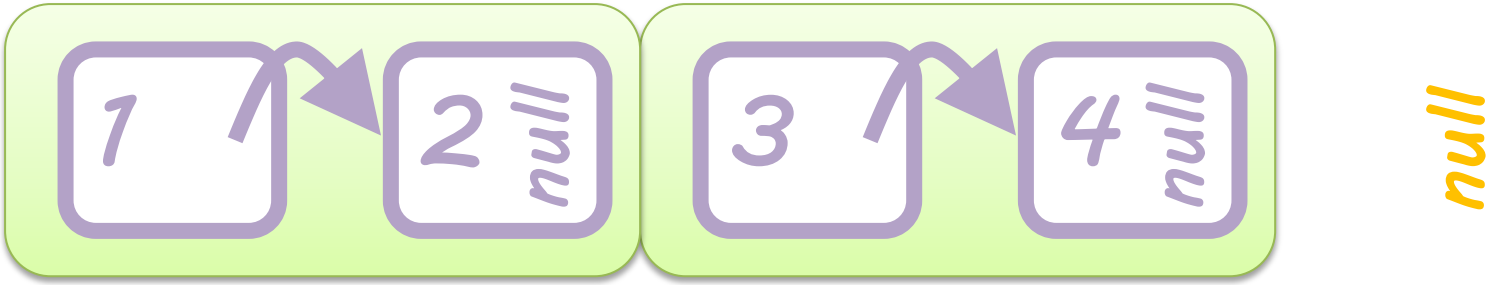


# (5) 小分けreverseをQueueで管理！



# 完成！

```
class Queue<T> {
  class Node { const T     value;
               const Node next; }
  const Queue<Node<T>> for_pop;
  const Node           for_push; }
```



# ここがCute!

- “Bootstrapping” と呼ばれています
  - OS を起動するための OS
  - コンパイラ をコンパイルするための コンパイラ
  - 同じことがデータ構造/アルゴリズムでも考えられる
    - Queue を実装するための Queue

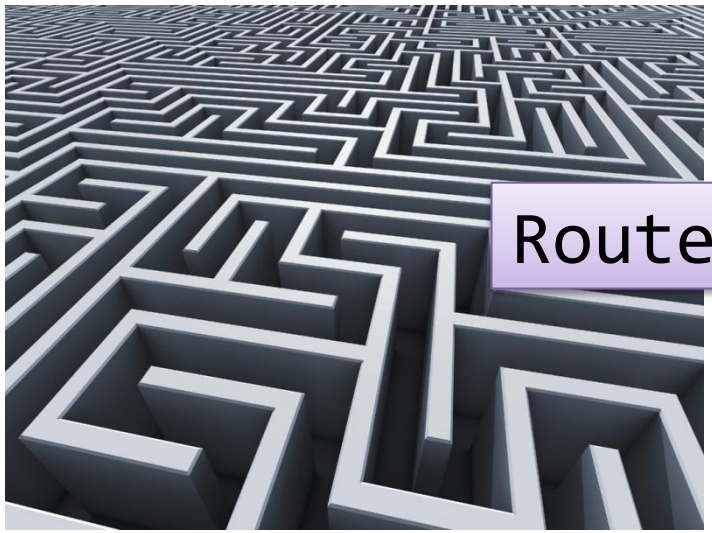
迷路でぶーとすとらっぴんぐ

# BOOTSTRAPPING IN A MAZE

参考：http://www.kmonos.net/wlog/105.html#_2232100114

# よくあるアルゴリズムの問題

与えられた迷路を通り抜ける
最短ルートを求めるプログラムを書きなさい



```
Route shortest(Maze m);
```

# ちょっと話を膨らませましょう

与えられた迷路を通り抜ける
最短ルートを求めるプログラムを書きなさい

最短であることの検査ルーチンも書きなさい

```
Route shortest(Maze m);
bool check(Maze m, Route r);
Route safe_shortest(Maze m) {
 Route r = shortest(m);
 if(!check(m,r)) throw Error;
 return r;
}
```

# 失敗例：意味のないチェック

shortest が仮に間違っていても
shortest の結果に*true*を返してしまう

```
Route shortest(Maze m);
bool check(Maze m, Route r) {
  return r.len == shortest(m).len;
}
```

– とはいえ、最短であることをチェックするには
最短路を計算するしかなさそう…？

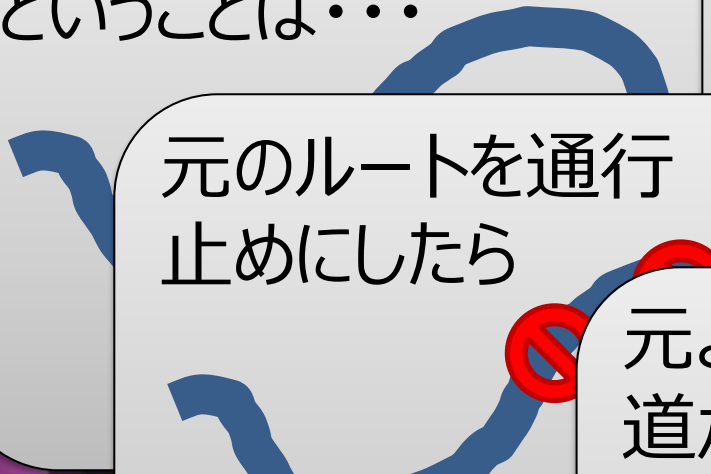
# Bootstrapping!

元の迷路をそのままshortestに投げるとダメ。
「小さい」迷路を渡すようにすれば！

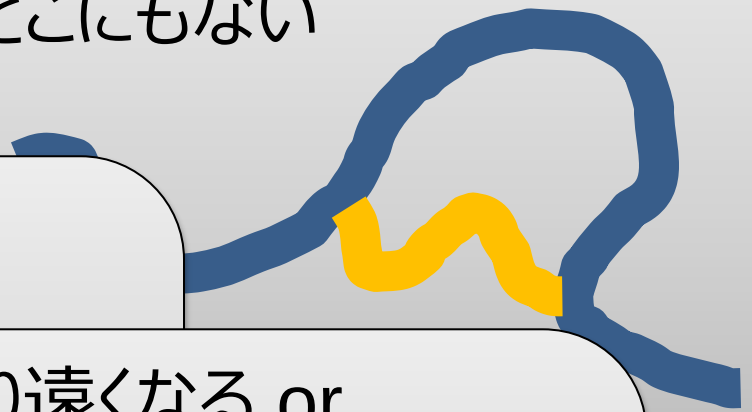
```
Route safe_shortest(Maze m);

bool check(Maze m, Route r) {
  // return r == safe_shortest(m);
  もっと“巧く”shortestを使う!!
}
```

# 最短ルート＝他の道は遠回り

これが最短
ということは・・・



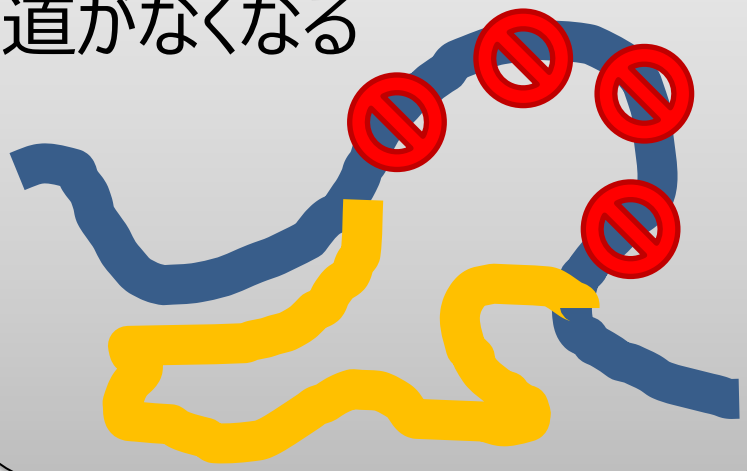
こういう近道が
どこにもない



元のルートを通行
止めにしたら



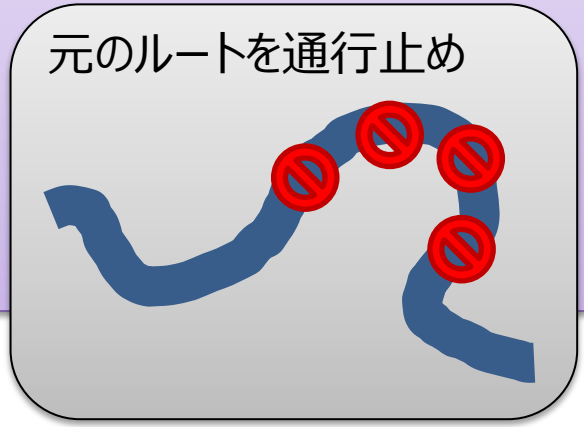
元より遠くなる or
道がなくなる



# Bootstrapping!

（※詳細略）（※条件略）

```
bool check(Maze m, Route r) {
 for( p,q ← r上の場所のペア ) {
  m’ = mのpからqまでを全て埋め立てた迷路
   if( safe_shortest(m’).len < r.len )
    return false;
 }
 return true;
}
```



# ここがCute!

- 間違ってるかもしれないルーチンを、
  元のルーチンの実装によらずに ※注
  間違いを自分で検出するルーチンに強化！

※注： 今回の手法は、「最短じゃなくてもルートがある時は少なくとも何かルートを返す」ことを暗黙に仮定しています。

正確な実数

# EXACT REALS

出典: “RealLib”, Barnimir Lambov, 2007-

# “誤差なし” 実数計算

http://wayback.archive.org/web/*/www.brics.dk/~barnie/RealLib/

```
using namespace RealLib;
int main() {
  Real sq3 = cos(Pi/3-Pi/2)*2;
  Real x   = 1;
  for(int i=0; i<4; ++i) x *= sq3;

  cout << setprecision(5) << x;
  // 9.00000
```

# “誤差なし” 実数計算

http://wayback.archive.org/web/*/www.brics.dk/~barnie/RealLib/

```
    cout << setprecision(50) << x;
    // 9.00000000000000000000000000000000
    // 00000000000000000000000000
    cout << setprecision(99999) << x;
    // 9.00000000000000000000000000000000
    // 00000000000000000000000000000000
    // 0000000000000.....000
}
```

# “BigDecimal”との違い

```
getcontext().prec = 80
x = Decimal(8)/Decimal(7)*Decimal(7)
print(x)
# Decimal(“7.99999....99997”)
```

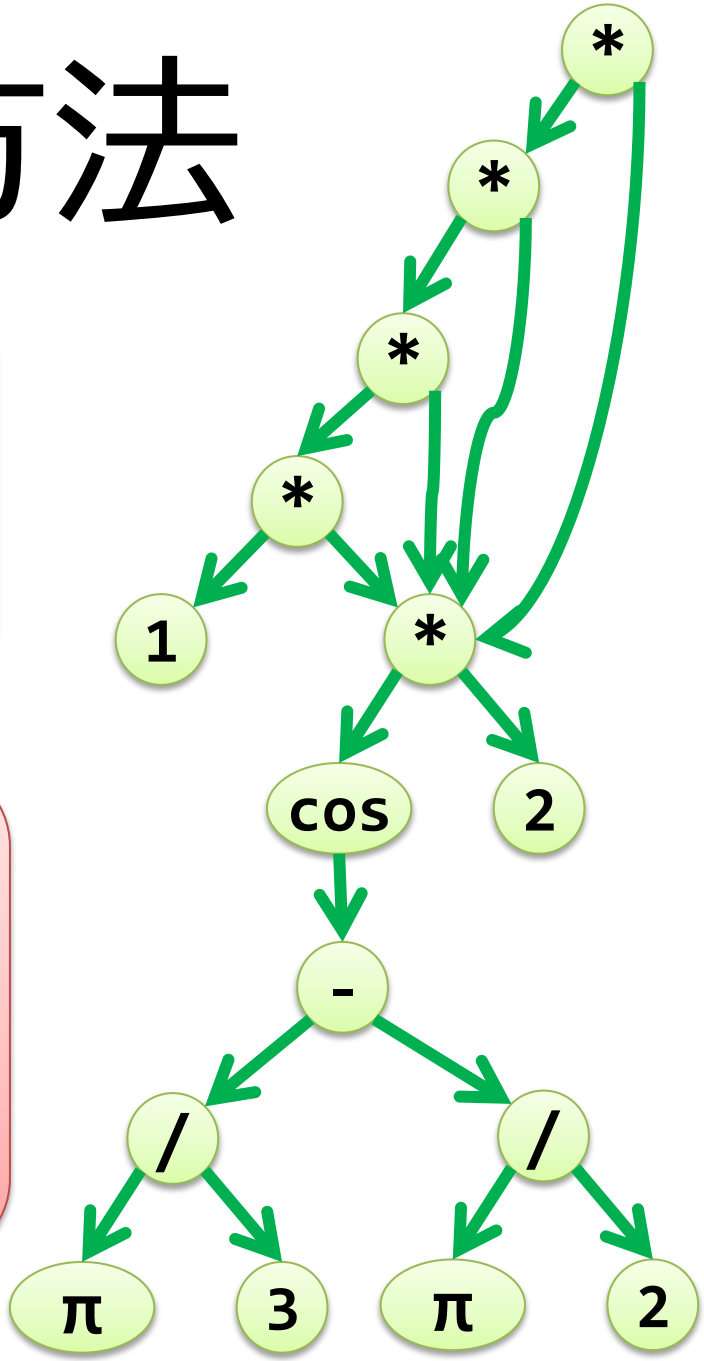
- 精度を最初に指定しなくていい
- いつでも好きな精度まで正確な値が得られる

```
Real x = Real(8)/Real(7)*Real(7)
cout << setprecision(80) << x;
// 8.00000....00000
```

# 実装方法

```
Real sq3 = cos(Pi/3-Pi/2)*2;
Real x   = 1;
for(int i=0; i<4; ++i)
  x *= sq3;
```

*9·00* でも
*9·00000000* でも
*9·00000…00000* でもなく
変数に「数式」をそのまま記憶する
（表示の時に必要精度まで計算）



# ここがCute!

- **シンプル！**
- こんなのありなの？と思ってしまうくらい
  言われて見れば当たり前
- でも効果は抜群

※ 同じ効果は「無限精度で計算する処理」
を「遅延評価」することでも実現できます

式のまま計算：その２

# EXPRESSION TREE

参考：http://www.kmonos.net/pub/files/sred_full.pdf

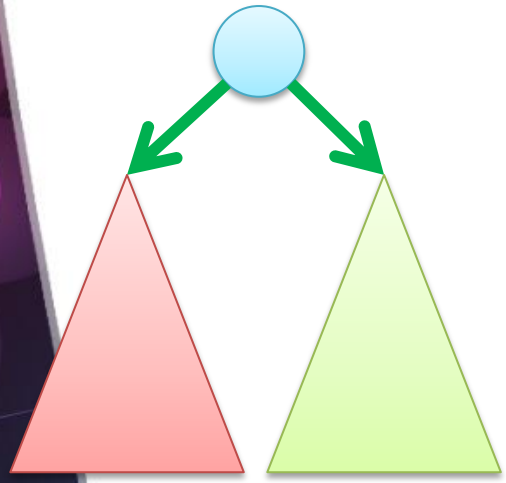
# Tree に対する Query 言語

```
SELECT (x, y) WHERE
 x in a AND
 y in * AND
 x.href = y.id
 AND
 p > LCA(x, y)
```

```
<html>...<p>...
  <a href="#xx"/>
  <img id="xx"/>
  ...
...</p>...</html>
```

# 実装（概要）

```
SELECT (x, y)
 x in a AND
 y in * AND
 x.href = y.id
 AND p > LCA(x, y)
```



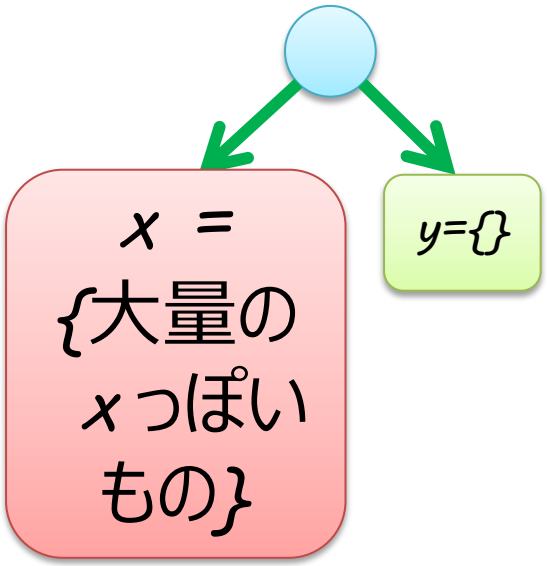
```
Result query(Tree t) {
 L = query(t.left)
 R = query(t.right)
 x   = L.x ∪ R.x ∪
       t.label="a" ? {t} : {}
 y   = ...
 xy  = {(a,b) | a∈x, b∈y,...}
 ans = L.ans ∪ R.ans ∪
       t.label="p" ? xy : {}
 return {x,y,xy,ans}
}
```

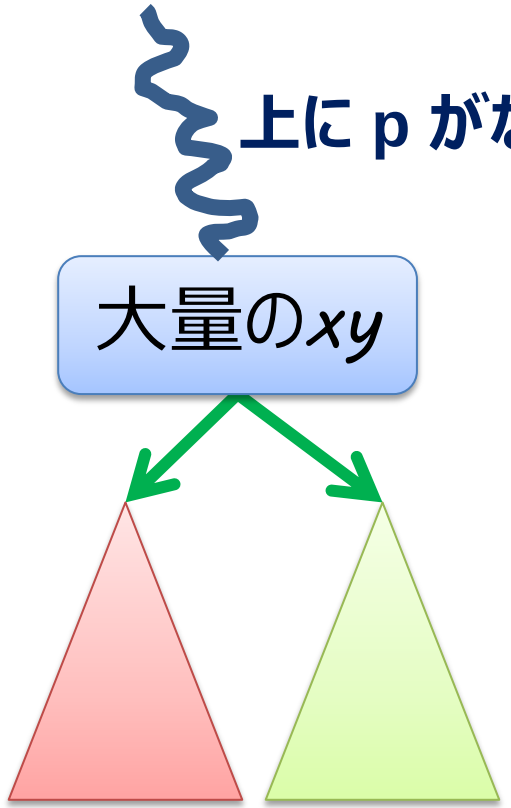
**左と右の「xっぽいもの」と**
**「yっぽいもの」を組み合わせ、を繰り返す**

# 問題点

「最終的に無駄」
な計算をたくさんしてしまう

```
SELECT (x, y)
 x in a AND
 y in * AND
 x.href = y.id
 AND p > LCA(x, y)
```



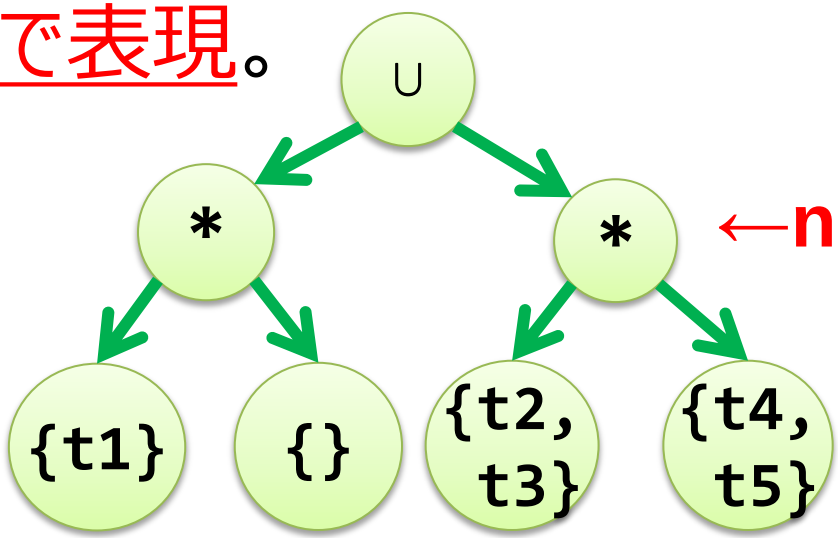
# かっこよくない解決策

*query*関数の構造を変えて、
木の先祖ノードや、兄弟をたどる、
「後で無駄にならないか？」の判定を足す

```
Result query(Tree t, INFO inf) {
 if( ... inf ... ) return
 inf = update_info(L)
 inf = update_info(R)
 if(...) L = query(t.left, inf)
 if(...) R = query(t.right, inf)
 if(...) x=... else x=...
}
```

# かっこいい解決策

集合をいきなり値で計算するのではなく、
計算式で表現。




←not: {(t2, t4), (t3, t4), (t2, t5), (t3, t5)}

- 単に使われない「実は無駄」な集合は、計算する前に捨てられる
- 空集合との掛け算は、あとで取り除く

# ここがCute!

- アルゴリズムを一切変えないでよい

```
Result query(Tree t) {
 L = query(t.left)
 R = query(t.right)
 ...
```

- データ構造だけを差し替えればよい
- 簡単！

# Thank you for listening!

- cast -

Slowest Sort

Bootstrapping

Expression Tree