計算機科学概論演習

#06

青谷知幸

Tokyo Tech., Dept. of Math. & Comp. Sci.

May 18, 2016

目次

部分構造を扱わない再帰函数
- 分割統治法 – クイックソートを例として
- 最大公約数の計算 – ユークリッドの互除法

バックトラックを用いたアルゴリズム

蓄積 (accumulation) の利用
- 閉路のあるグラフにおける経路探索
- 函数の効率化

課題

部分構造を扱わない再帰函数の雛型

部分構造を扱う再帰函数よりも抽象的になるが，大体以下のように書くことになる

```
def f: T => R =
(arg) => {
  ... match{
    case 解が明らかな場合: ...
    case 小さな問題に分割する必要がある場合:
      ... f(genProblem_1(...)) ... f(genProblem_2(...)) ...
      ... f(genProblem_n(...)) ...
  }
}
```

部分構造を扱わない再帰函数の雛型

部分構造を扱う再帰函数よりも抽象的になるが，大体以下のように書くことになる

```
def f: T => R =
(arg) => {
  if(解が明らかな場合){...}
  else{
      ... f(genProblem_1(...)) ... f(genProblem_2(...)) ...
      ... f(genProblem_n(...)) ...
  }
}
```

再帰函数の雛型の比較

部分構造を扱う再帰函数

```
def f: List[T] => R =
(lst) => {
  lst match{
    case Nil:
      ...
    case e::rst:
     ...
     f(rst)
     ...
  }
}
```

部分構造を扱わない再帰函数

```
def f: T => R =
(arg) => {
  ... match{
    case 解が明らか:
      ...
    case 小さな問題に分割:
     ...
     f(genProblem_1(...))
     ...
     f(genProblem_n(...))
     ...
  }}
```

部分構造を扱わない再帰函数は扱う再帰函数の一般形

部分構造を扱わない再帰函数の注意点: 停止性

部分構造を扱う再帰函数に比べて部分構造を扱わない再帰函数は停止することの確認が難しい

部分構造を扱わない再帰函数の注意点: 停止性

部分構造を扱う再帰函数に比べて部分構造を扱わない再帰函数は停止することの確認が難しい

- 部分構造を扱う再帰函数: 再帰呼出しを繰り返して解が明らかな場合に辿り着くことは自明

部分構造を扱わない再帰函数の注意点: 停止性

部分構造を扱う再帰函数に比べて部分構造を扱わない再帰函数は停止することの確認が難しい

- 部分構造を扱う再帰函数: 再帰呼出しを繰り返して解が明らかな場合に辿り着くことは自明
  - 部分リストを扱う再帰函数: 再帰呼出しのたびにリストの長さが 1 ずつ短くなる $\Rightarrow$ いつかは `Nil` に辿り着いて停止

部分構造を扱わない再帰函数の注意点: 停止性

部分構造を扱う再帰函数に比べて部分構造を扱わない再帰函数は停止することの確認が難しい

- 部分構造を扱う再帰函数: 再帰呼出しを繰り返して解が明らかな場合に辿り着くことは自明
  - 部分リストを扱う再帰函数: 再帰呼出しのたびにリストの長さが 1 ずつ短くなる $\Rightarrow$ いつかは `Nil` に辿り着いて停止
  - 1 小さい自然数を扱う再帰函数: 再帰呼出しのたびに値が 1 ずつ小さくなる $\Rightarrow$ いつかは 0 に辿り着いて停止

部分構造を扱わない再帰函数の注意点: 停止性

部分構造を扱う再帰函数に比べて部分構造を扱わない再帰函数は停止することの確認が難しい

- 部分構造を扱う再帰函数: 再帰呼出しを繰り返して解が明らかな場合に辿り着くことは自明
  - 部分リストを扱う再帰函数: 再帰呼出しのたびにリストの長さが 1 ずつ短くなる $\Rightarrow$ いつかは `Nil` に辿り着いて停止
  - 1 小さい自然数を扱う再帰函数: 再帰呼出しのたびに値が 1 ずつ小さくなる $\Rightarrow$ いつかは 0 に辿り着いて停止
- 部分構造を扱わない再帰函数: 再帰呼出しを繰り返して解が明らかな場合に辿り着けるかどうかは非自明

部分構造を扱わない再帰函数の注意点: 停止性

部分構造を扱う再帰函数に比べて部分構造を扱わない再帰函数は停止することの確認が難しい

- 部分構造を扱う再帰函数: 再帰呼出しを繰り返して解が明らかな場合に辿り着くことは自明
  - 部分リストを扱う再帰函数: 再帰呼出しのたびにリストの長さが 1 ずつ短くなる $\Rightarrow$ いつかは `Nil` に辿り着いて停止
  - 1 小さい自然数を扱う再帰函数: 再帰呼出しのたびに値が 1 ずつ小さくなる $\Rightarrow$ いつかは 0 に辿り着いて停止
- 部分構造を扱わない再帰函数: 再帰呼出しを繰り返して解が明らかな場合に辿り着けるかどうかは非自明
  $\Leftarrow$ 再帰呼出しするたびに “何らかの尺度” で解が自明な場合へ近づくことを調べるのが一般的

分割統治法: クイックソート

- 分割統治法: 問題を独立な小さな問題に分解して解くことで解を求める計算方法

分割統治法: クイックソート

- 分割統治法: 問題を独立な小さな問題に分解して解くことで解を求める計算方法
- クイックソート: 分割統治による高速な整列法
  compare を比較函数，lst を並び替えの対象のリスト，qsort をクイックソート函数とする．lst の先頭要素が e，残りのリストが rst であるとき，rst を
  - rstL≜List(x|∀x.x∈rst∧compare(x)(e)): rst の要素の x で，compare(x)(e) が真になるものばかりからなるリスト
  - rstR≜List(x|∀x.x∈rst∧!compare(x)(e)): rst の要素の x で，compare(x)(e) が偽になるものばかりからなるリスト

  に分解して，rstL と rstR についてそれぞれ qsort で整列した後，結果をつなぎ合わせる

qsort の振舞

qsort(lt)(List(3,2,4,1))

- qsort[X]: (X=>X=>Boolean)=>List[X]=>List[X]
- def lt: Int=>Int=>Boolean = (l)=>(r)=>{l < r}

qsort(lt)(lst) のやること

lst が e::rst のとき，rst を

- rstL≜List(x|∀x.x∈rst∧lt(x)(e)): rst の要素の x で，lt(x)(e) が真になるものばかりからなるリスト
- rstR≜List(x|∀x.x∈rst∧!lt(x)(e)): rst の要素の x で，lt(x)(e) が偽になるものばかりからなるリスト

に分解して，rstL と rstR についてそれぞれ qsort で整列した後，結果をつなぎ合わせる

qsort の振舞

qsort(lt)(List(3,2,4,1))

$=$ //e$\triangleq$3, rst$\triangleq$List(2,4,1), rstL$\triangleq$List(2,1), rstR$\triangleq$List(4)

qsort(lt)(List(2,1)) ++ (3 :: qsort(lt)(List(4)))

qsort の振舞

qsort(lt)(List(3,2,4,1))

$=$ //e$\triangleq$3, rst$\triangleq$List(2,4,1), rstL$\triangleq$List(2,1), rstR$\triangleq$List(4)

qsort(lt)(List(2,1)) ++ (3 :: qsort(lt)(List(4)))

$=$ //e$\triangleq$2, rst$\triangleq$List(1), rstL$\triangleq$List(1), rstR$\triangleq$Nil

qsort(lt)(List(1)) ++ (2 :: qsort(lt)(Nil)) ++
(3 :: qsort(lt)(List(4)))

qsort の振舞

```
qsort(lt)(List(3,2,4,1))
= //e≜3, rst≜List(2,4,1), rstL≜List(2,1), rstR≜List(4)
qsort(lt)(List(2,1)) ++ (3 :: qsort(lt)(List(4)))
= //e≜2, rst≜List(1), rstL≜List(1), rstR≜Nil
qsort(lt)(List(1)) ++ (2 :: qsort(lt)(Nil)) ++
(3 :: qsort(lt)(List(4)))
=//e≜1, rst≜Nil, rstL≜Nil, rstR≜Nil
qsort(lt)(Nil) ++ (1 :: qsort(lt)(Nil)) ++
(2 :: qsort(lt)(Nil)) ++ (3 :: qsort(lt)(List(4)))
```

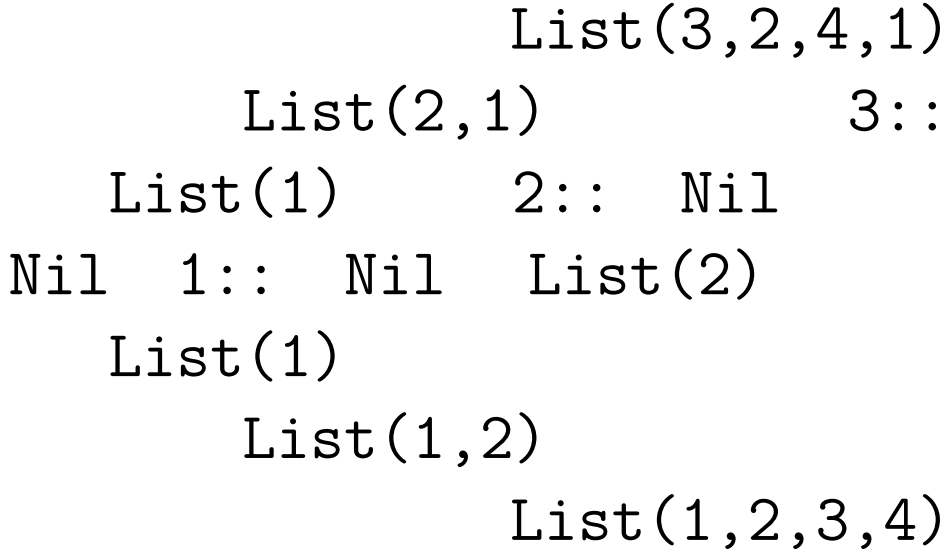
qsort の振舞

```
qsort(lt)(List(3,2,4,1))
= //e≜3, rst≜List(2,4,1), rstL≜List(2,1), rstR≜List(4)
qsort(lt)(List(2,1)) ++ (3 :: qsort(lt)(List(4)))
= //e≜2, rst≜List(1), rstL≜List(1), rstR≜Nil
qsort(lt)(List(1)) ++ (2 :: qsort(lt)(Nil)) ++
(3 :: qsort(lt)(List(4)))
=//e≜1, rst≜Nil, rstL≜Nil, rstR≜Nil
qsort(lt)(Nil) ++ (1 :: qsort(lt)(Nil)) ++
(2 :: qsort(lt)(Nil)) ++ (3 :: qsort(lt)(List(4)))
=
Nil ++ List(1) ++ List(2) ++ List(3,4)
```

qsort の振舞

```
qsort(lt)(List(3,2,4,1))
= //e≜3, rst≜List(2,4,1), rstL≜List(2,1), rstR≜List(4)
qsort(lt)(List(2,1)) ++ (3 :: qsort(lt)(List(4)))
= //e≜2, rst≜List(1), rstL≜List(1), rstR≜Nil
qsort(lt)(List(1)) ++ (2 :: qsort(lt)(Nil)) ++
(3 :: qsort(lt)(List(4)))
=//e≜1, rst≜Nil, rstL≜Nil, rstR≜Nil
qsort(lt)(Nil) ++ (1 :: qsort(lt)(Nil)) ++
(2 :: qsort(lt)(Nil)) ++ (3 :: qsort(lt)(List(4)))
=
Nil ++ List(1) ++ List(2) ++ List(3,4)
=
List(1,2,3,4)
```

qsort(lt)(List(3,2,4,1))



lst の先頭要素が e，残りのリストが rst であるとき，rst を

- rstL≜List(x|∀x.x∈rst∧p(x)(e)): rst の要素の x で，p(x)(e) が真になるものばかりからなるリスト
- rstR≜List(x|∀x.x∈rst∧!p(x)(e)): rst の要素の x で，p(x)(e) が偽になるものばかりからなるリスト

に分解し，rstL と rstR を qsort で整列してつなぎ合わせる

qsort: コード例

```
def qsort[X]: (X=>X=>Boolean) => List[X] => List[X]
= (compare) => (lst) =>
/**
 * 目的: compareに従ってlstの要素を整列する
 * 例: qsort(lt)(List(3,2,4,1)) == List(1,2,3,4)
 *     qsort(gt)(List(3,2,4,1)) == List(4,3,2,1)
 */
{...}

def lt: Int => Int => Boolean = (l) => (r) => {l < r}
def gt: Int => Int => Boolean = (l) => (r) => {l > r}
```

qsort: コード例

```
def qsort[X]: (X=>X=>Boolean) => List[X] => List[X]
= (compare) => (lst) =>
/**
 * 目的: compareに従ってlstの要素を整列する
 * 例: qsort(lt)(List(3,2,4,1)) == List(1,2,3,4)
 *     qsort(gt)(List(3,2,4,1)) == List(4,3,2,1)
 */
lst match{
  case Nil => Nil
  case e::rst => //rstL≜List(x|∀x.x∈rst∧compare(x)(e)) を整列
    qsort(compare)( rstL ) ++
    (e::
     qsort(compare)( rstR )
    )          //rstR≜List(x|∀x.x∈rst∧!compare(x)(e)) を整列
}
```

qsort: コード例

```
def qsort[X]: (X=>X=>Boolean) => List[X] => List[X]
= (compare) => (lst) => lst match{
  case Nil => Nil
  case e::rst => //rstL≜List(x|∀x.x∈rst∧compare(x)(e)) を整列
    qsort(compare)(filter(flip(compare)(e))(rst)) ++
    (e::
     qsort(compare)( rstR )
    )              //rstR≜List(x|∀x.x∈rst∧!compare(x)(e)) を整列
}

def flip[X,Y,Z]: (X=>Y=>Z) => Y => X => Z =
(f) => (r) => (l) =>
/**
 * 第一引数と第二引数を入れ替える: flip(lt)(3)(4) == lt(4)(3)
 */
{f(l)(r)}
```

qsort: コード例

```
def qsort[X]: (X=>X=>Boolean) => List[X] => List[X]
= (compare) => (lst) => lst match{
  case Nil => Nil
  case e::rst => //rstL≜List(x|∀x.x∈rst∧compare(x)(e)) を整列
    qsort(compare)(filter(flip(compare)(e))(rst)) ++
    (e::
     qsort(compare)(filter(flip(negate(compare))(e))(rst))
    )            //rstR≜List(x|∀x.x∈rst∧!compare(x)(e)) を整列
}

def negate[X]: (X=>X=>Boolean) => X => X => Boolean =
(p) => (l) => (r) => {!p(l)(r)}
```

qsort の停止性

qsort の第二引数は再帰呼出しのたびに長さが短くなる (部分リストではないが). 従って再帰呼出しを繰り返すと Nil に辿り着き，停止する.

```
def qsort[X]: (X=>X=>Boolean) => List[X] => List[X]
= (compare) => (lst) => lst match{
  case Nil => Nil
  case e::rst => //rstL≜List(x|∀x.x∈rst∧compare(x)(e)) を整列
    qsort(compare)(filter(flip(compare)(e))(rst)) ++
    (e::
     qsort(compare)(filter(flip(negate(compare))(e))(rst))
    )              //rstR≜List(x|∀x.x∈rst∧!compare(x)(e)) を整列
}
```

最大公約数を求める

問題
n, m∈Int が与えられた時，n と m の最大公約数を求めるプログラム gcd を開発せよ．

解法
部分構造を扱う再帰函数による解法と，部分構造を扱わない再帰函数による解法が考えられる．

部分構造を扱う解法

n と m の大きくない方を i とする．i==1 になるまで i を 1 ずつ小さくしながら n%i==0 かつ m%i==0 となる i をさがす．

```
def gcd: Int => Int => Int = (n) => (m) => {
  def find: Int => Int = (i) => i match{
    case 1 => 1
    case _ => if(n%i==0 && m%i==0) i else find(i-1)
  }
  find(min(n)(m))
}

def min: Int => Int => Int
= (n) => (m) => {if(n<m) n else m}
```

部分構造を扱わない解法

n と m の大きくない方を smaller，そうでない方を another とする．smaller が 0 になるまで，smaller を another%smaller，another を smaller として計算を繰り返す．smaller が 0 になった時，another が解である．

```
def gcd: Int => Int => Int = (n) => (m) => {
  def find: Int => Int => Int = (smaller) => (another) =>
  smaller match{
    case 0 => another
    case _ => find(another%smaller)(smaller)
  }          //0≤another%smaller<smaller⇒いつかは停止
  find(min(n)(m))(max(n)(m))
}

def max: Int => Int => Int
= (n) => (m) => {if(n<m) m else n}
```

目次

部分構造を扱わない再帰函数
- 分割統治法 – クイックソートを例として
- 最大公約数の計算 – ユークリッドの互除法

バックトラックを用いたアルゴリズム

蓄積 (accumulation) の利用
- 閉路のあるグラフにおける経路探索
- 函数の効率化

課題

概要

　迷宮を探索する時，出口に真っ直ぐに辿り着くということはあまり期待できない．とりあえず方向を決めて探索を進め，出口が見つからなかった時，探索の途中の時点まで戻ってから，別の方向に探索を進める．

　同様の手法が，ある種の問題を解く際にも用いられる．今回は 2 種類の問題を通して「途中まで戻りながら」（バックトラック）探索を続行するような問題の解き方を学ぶ．

- グラフ探索
- チェスに関する問題 (課題)

グラフ探索と有向グラフ

グラフ探索は一般的な探索問題である．

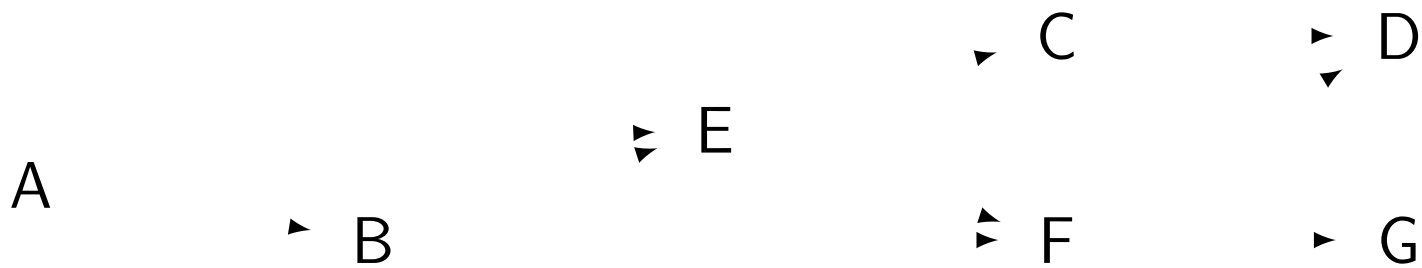
- 一方通行の通路で構成された迷宮を自動生成した．宝物を取って地上に出る路が存在することを確認するプログラムを開発せよ．
- 鉄道網を表すグラフが与えられた．家の最寄り駅から空港駅へ鉄道で往く際の乗り換え方法を検索するプログラムを開発せよ．
  - 通常は「一番安い運賃」や「最短時間」などの制約が入るが，ここでは簡単のために一つでも見つかればよいものとしよう

このような問題において，迷宮や鉄道網を表すために有向グラフ (directed graph)という道具を用いる．

有向グラフの定義

Vを頂点 (nodes) の集合，Eを辺 (edges) の集合とする．1 本の辺は 2 つの頂点を繋ぐものである．従って $E \subseteq V \times V$である．辺は一方向にしか辿ることが出来ない．

有向グラフの例

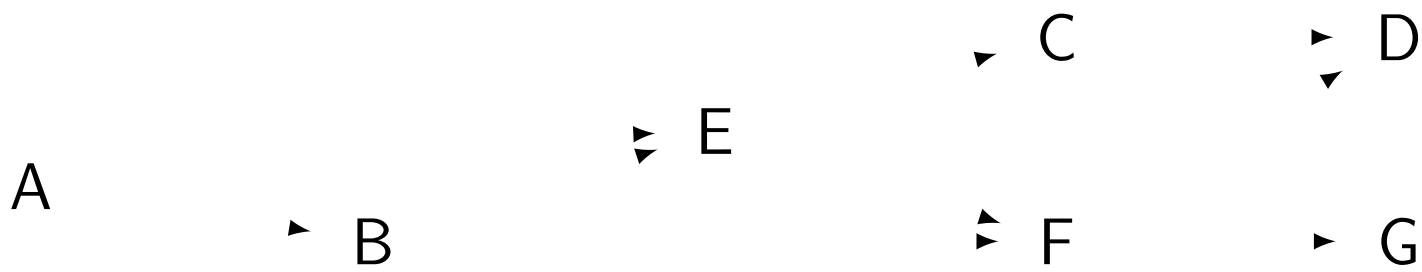


白い円は頂点を，矢印は頂点から頂点への一方通行の接続 (有向グラフの辺) を表している．$|V| = 7, |E| = 9$である．

グラフ探索: 例題

以下の有向グラフについて，以下を満たす経路を答えよ

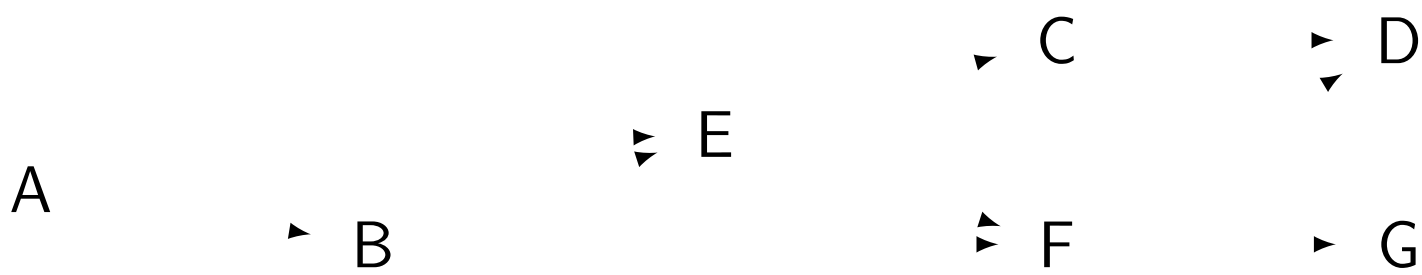
1. 頂点 C から頂点 D:
2. 頂点 E から頂点 D:
3. 頂点 C から頂点 G:



グラフ探索: 例題

以下の有向グラフについて，以下を満たす経路を答えよ

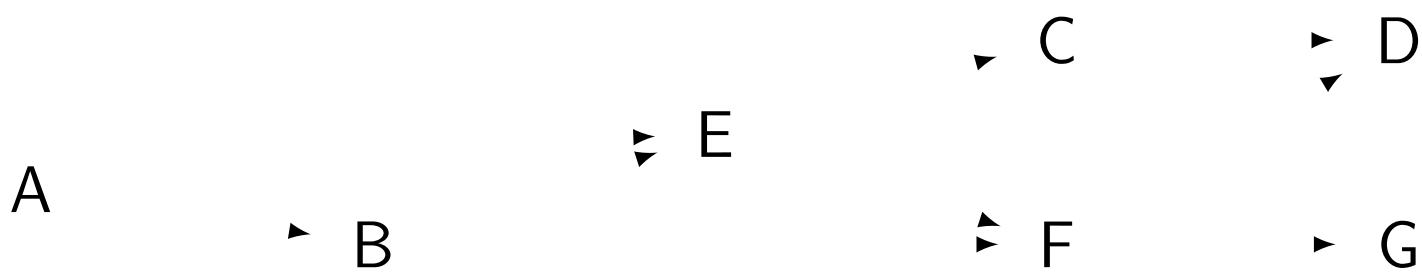
1. 頂点 C から頂点 D: C→D
2. 頂点 E から頂点 D:
3. 頂点 C から頂点 G:



グラフ探索: 例題

以下の有向グラフについて，以下を満たす経路を答えよ

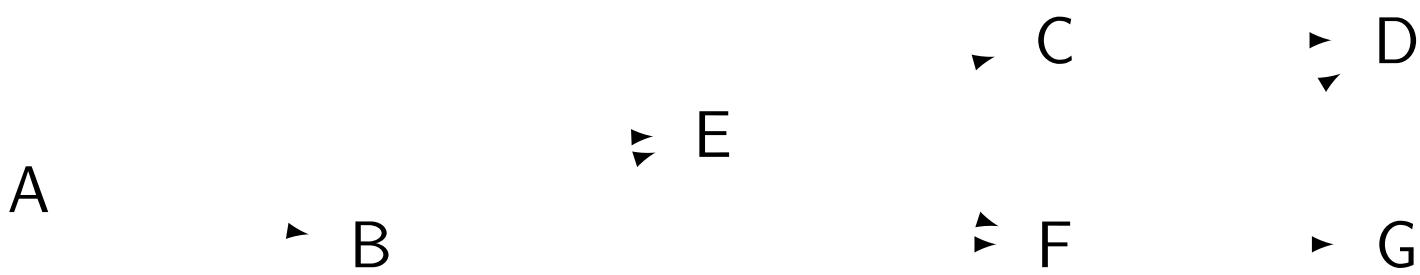
1. 頂点 C から頂点 D: C→D
2. 頂点 E から頂点 D: E→F→D または E→C→D
3. 頂点 C から頂点 G:



グラフ探索: 例題

以下の有向グラフについて，以下を満たす経路を答えよ

1. 頂点 C から頂点 D: C→D
2. 頂点 E から頂点 D: E→F→D または E→C→D
3. 頂点 C から頂点 G: 解なし



グラフを表すデータ構造

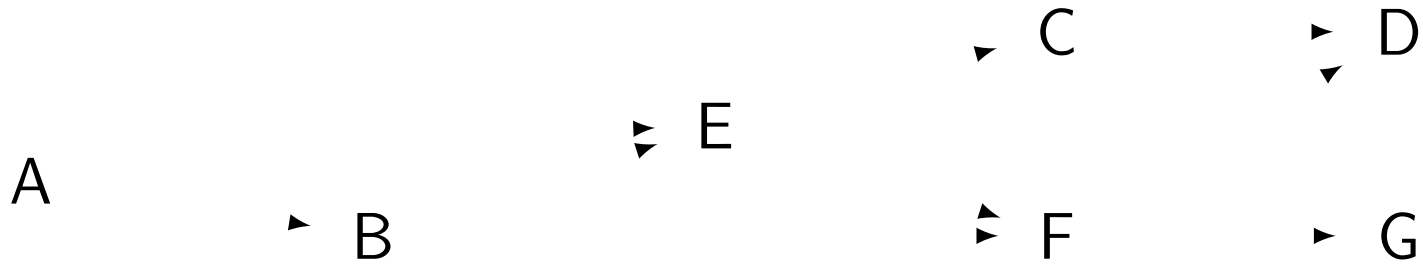
有向グラフを表すデータ構造として，各頂点がどの頂点と辺でつながっているかを表現するリストを用いる

- 頂点: `Symbol`
- グラフ: `List[Conn]`

```
case class Conn(node: Symbol, neighbors: List[Symbol])
```

例: グラフの表現

```
List(Conn('A, List('B,'E)), Conn('B, List('E,'F)),
     Conn('C, List('D)), Conn('D, Nil), Conn('E, List('C, 'F)),
     Conn('F, List('D, 'G)), Conn('G, Nil))
```



探索函数 findRoute

findRoute は始点と終点とグラフが与えられた時に，始点から終点までの経路を返す函数である．経路は訪れる順番に頂点を並べたリスト (List[Symbol]) で表現する．経路がない場合は空のリストで表現する．

```
def findRoute: (Symbol, Symbol, List[Conn]) => List[Symbol]
=
(origination, destination, graph) =>
{
  if(解が自明に求まる) ...
  else ...
}
```

解が自明に求まるのはどういう時か？

探索函数 findRoute

始点と終点が同じ時，解は自明である．ただ 1 頂点のみの経路である．

```
def findRoute: (Symbol, Symbol, List[Conn]) => List[Symbol]
= (origination, destination, graph) =>
{
  if(origination == destination) List(destination)
  else ...
}
```

解が自明に求まらない時は，部分問題を生成してそれを解く．
部分問題は何か？

探索函数 findRoute

部分問題は，始点から辺を辿って隣の頂点に移った時，そこを始点として終点までの経路を求めることである．

```
def findRoute: (Symbol, Symbol, List[Conn]) => List[Symbol]
= (origination, destination, graph) => {
    if(origination == destination) List(destination)
    else findRouteL(neighborsOf(origination,graph),
                    destination, graph) match{
      case Nil => Nil
      case ans => origination::ans
    }}
```

neighborsOf および findRouteL は補助函数である．

補助函数

```
def neighborsOf: (Symbol, List[Conn]) => List[Symbol]
=
(node, graph) =>
/**
 * 目的: nodeの隣の頂点を返す
 */
{...}

def findRouteL: (List[Symbol], Symbol, List[Conn])
                => List[Symbol]
=
(originations, destination, graph) =>
/**
 * 目的: originationsからdestinationに辿り着く経路を求める
 */
{...}
```

補助函数 neighborsOf

```
def neighborsOf: (Symbol, List[Conn]) => List[Symbol]
=
  (node, graph) => {
    def eqNode: Conn => Boolean
    = (c) => c match{ case Conn(n, _) => node == n }

    find(eqNode, graph) match{
      case None => Nil
      case Some(c) => c match{ case Conn(_,n) => n }
    }}
```

ただし

- `find` は補助函数
- `Option[X]` はリストと同じく scala に用意されている基本的なデータ構造の 1 つである.

補助函数 neighborsOf

```
abstract class Option[+X]
case class Some[+X](x: X) extends Option[X]
case object None extends Option[Nothing]

def find[X]: ((X) => Boolean, List[X]) => Option[X]
=
  (p, lst) => lst match{
    case Nil => None
    case hd::tl => if(p(hd)) Some(hd) else find(p, tl)
  }
```

補助函数 neighborsOf

```
def neighborsOf: (Symbol, List[Conn]) => List[Symbol]
=
  (node, graph) => {
    def eqNode: Conn => Boolean
    = (c) => c match{ case Conn(n, _) => node == n }

    find(eqNode, graph) match{
      case None => Nil
      case Some(c) => c match{ case Conn(_,n) => n }
    }}
```

ただし

- `find` は補助函数
- `Option[X]` はリストと同じく scala に用意されている基本的なデータ構造の 1 つである.

補助函数

```
def neighborsOf: (Symbol, List[Conn]) => List[Symbol]
=
(node, graph) =>
/**
 * 目的: nodeの隣の頂点を返す
 */
{...}

def findRouteL: (List[Symbol], Symbol, List[Conn])
                => List[Symbol]
=
(originations, destination, graph) =>
/**
 * 目的: originationsからdestinationに辿り着く経路を求める
 */
{...}
```

補助函数 findRouteL

findRouteL は originations が

- 空の時: Nil を返す
- 空でない時: 先頭要素である頂点から destination までの経路を findRoute によって求め，
  - 経路が見つかった時: その経路を返す
  - 経路が見つからない時: originations の残りの頂点から destination までの経路を findRouteL によって求める
    ⇐ それまでの探索を捨てて新たに探索を再開している
    ≙ バックトラック

補助函数 findRouteL

findRouteL は originations が

- 空の時: Nil を返す
- 空でない時: 先頭要素である頂点から destination までの経路を findRoute によって求め,
  - 経路が見つかった時: その経路を返す
  - 経路が見つからない時: originations の残りの頂点から destination までの経路を findRouteL によって求める

```
def findRouteL: (List[Symbol], Symbol, List[Conn])
                  => List[Symbol]
= (originations, destination, graph) => originations match{
    case Nil => Nil
    case hd::tl => findRoute(hd, destination, graph) match{
      case Nil => findRouteL(tl, destination, graph)
      case ans => ans
    }}
```

findRoute の完成

```
def findRouteL: (List[Symbol], Symbol, List[Conn])
                => List[Symbol]
= (originations, destination, graph) => originations match{
    case Nil => Nil
    case hd::tl => findRoute(hd, destination, graph) match{
      case Nil => findRouteL(tl, destination, graph)
      case ans => ans
    }}

def findRoute: (Symbol, Symbol, List[Conn]) => List[Symbol]
= (origination, destination, graph) => {
    if(origination == destination) List(destination)
    else findRouteL(neighborsOf(origination,graph),
                    destination, graph) match{
      case Nil => Nil
      case ans => origination::ans
    }}
```

目次

部分構造を扱わない再帰函数
- 分割統治法 – クイックソートを例として
- 最大公約数の計算 – ユークリッドの互除法

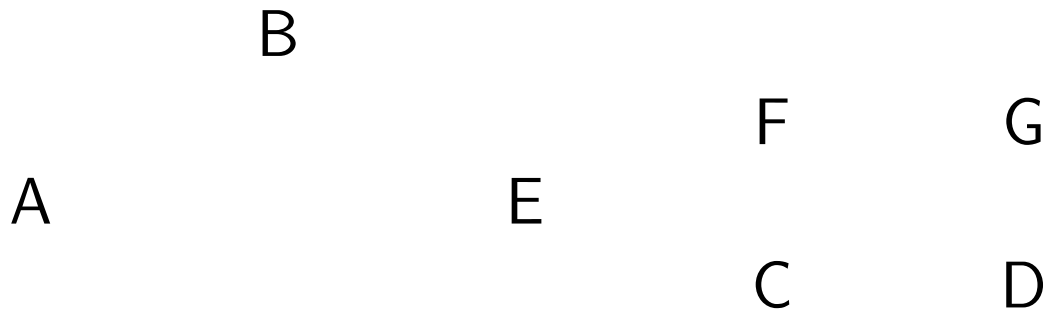
バックトラックを用いたアルゴリズム

蓄積 (accumulation) の利用
- 閉路のあるグラフにおける経路探索
- 函数の効率化

課題

閉路有りグラフの経路探索

グラフに閉路があると経路の探索は難しくなる

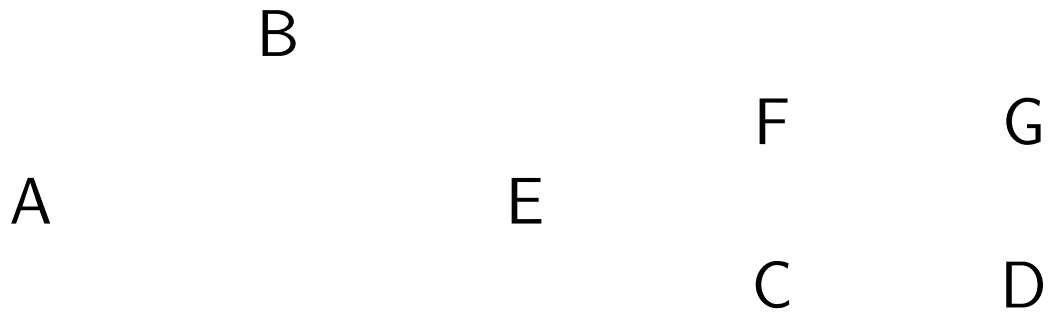


例:

$Q.$ A から D への行き方は何通りあるか？

閉路有りグラフの経路探索

グラフに閉路があると経路の探索は難しくなる



例:

$Q.$ A から D への行き方は何通りあるか？

$A.$ 無限

- $(\mathrm{A}\rightarrow \mathrm{E}\rightarrow \mathrm{C}\rightarrow)^{+}\ \mathrm{D}$

$\Rightarrow$ 経路探索プログラムが止まらなくなる

閉路有りグラフの経路探索

グラフに閉路がある時は，訪れた頂点を蓄積しながら探索すれば良い．これから訪れようとする頂点が

- 蓄積の中にある $\Rightarrow$ 他の頂点を辿る経路を探索
- 蓄積の中にない $\Rightarrow$ その頂点を辿る経路を探索

閉路有りグラフの経路探索

グラフに閉路がある時は，訪れた頂点を蓄積しながら探索すれば良い．これから訪れようとする頂点が

- 蓄積の中にある ⇒ 他の頂点を辿る経路を探索
- 蓄積の中にない ⇒ その頂点を辿る経路を探索

```
def findRouteAcc: (List[Symbol]) =>
                  (Symbol, Symbol, List[Conn]) =>
                  List[Symbol]
=
  (acc) => (orig, dest, graph) => {
    if(orig == dest) List(dest)
    else findRouteLAcc(orig::acc, neighborsOf(orig,graph),
                       dest, graph) match{
      case Nil => Nil
      case ans => orig::ans
    }}
```

閉路有りグラフの経路探索

グラフに閉路がある時は，訪れた頂点を蓄積しながら探索すれば良い．これから訪れようとする頂点が

- 蓄積の中にある $\Rightarrow$ 他の頂点を辿る経路を探索
- 蓄積の中にない $\Rightarrow$ その頂点を辿る経路を探索

```
def findRoute: (Symbol, Symbol, List[Conn]) => List[Symbol]
=
{findRouteAcc(Nil)} //訪問済みの頂点はない
```

閉路有りグラフの経路探索

グラフに閉路がある時は，訪れた頂点を蓄積しながら探索すれば良い．これから訪れようとする頂点が

- 蓄積の中にある ⇒ 他の頂点を辿る経路を探索
- 蓄積の中にない ⇒ その頂点を辿る経路を探索

```
def findRouteLAcc:
(List[Symbol],List[Symbol],Symbol,List[Conn]) => List[Symbol]
= (acc,origs,dest,graph) => origs match{
    case Nil => Nil
    case hd::tl => find(eqF(hd),acc) match{ //既にhdを訪問？
      case None =>                          //訪問していない
        findRouteAcc(hd::acc)(hd, dest, graph) match{
        case Nil => findRouteLAcc(acc, tl, dest, graph)
        case ans => ans
      }
      case _ => findRouteLAcc(acc, tl, dest, graph) //訪問済み
  }}
```

目次

部分構造を扱わない再帰函数
- 分割統治法 – クイックソートを例として
- 最大公約数の計算 – ユークリッドの互除法

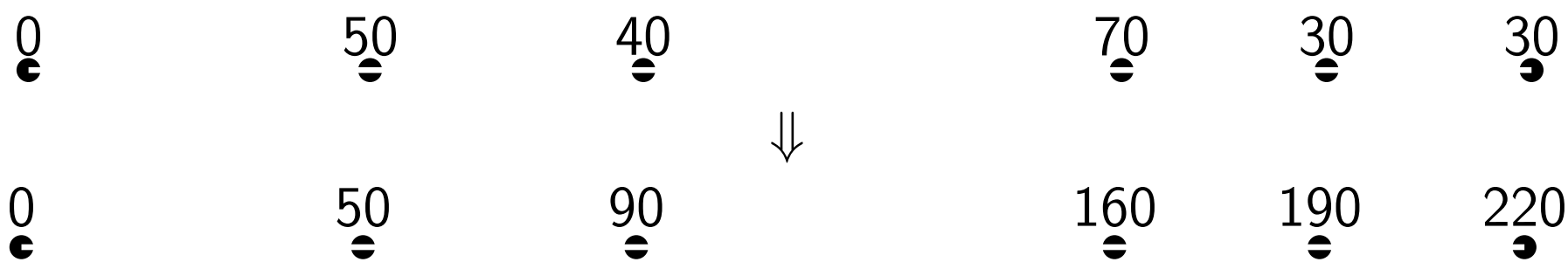
バックトラックを用いたアルゴリズム

蓄積 (accumulation) の利用
- 閉路のあるグラフにおける経路探索
- 函数の効率化

課題

例題 1: 相対距離 $\longrightarrow$絶対距離

問題
直線上に点 $p_0, p_1, \cdots, p_n$が並んでいる．r_iを p_{i-1}と p_iの距離 (ただし r_0は 0とする)，d_iを p_0と p_iの距離とする．
$\texttt{List}(r_0, \cdots, r_n)$ が与えられた時，これを $\texttt{List}(d_0, \cdots, d_n)$ に変換するプログラム r2a を開発せよ．



愚直な実装

```
def r2a: List[Int] => List[Int]
= {
  def aux: Int => List[Int] => List[Int]
  = (x) => (lst) => map(add(x))(0::lst);
  foldr(aux)(Nil)
}
```

愚直な実装

```
def r2a: List[Int] => List[Int]
= {
  def aux: Int => List[Int] => List[Int]
  = (x) => (lst) => map(add(x))(0::lst);
  foldr(aux)(Nil)
}
```

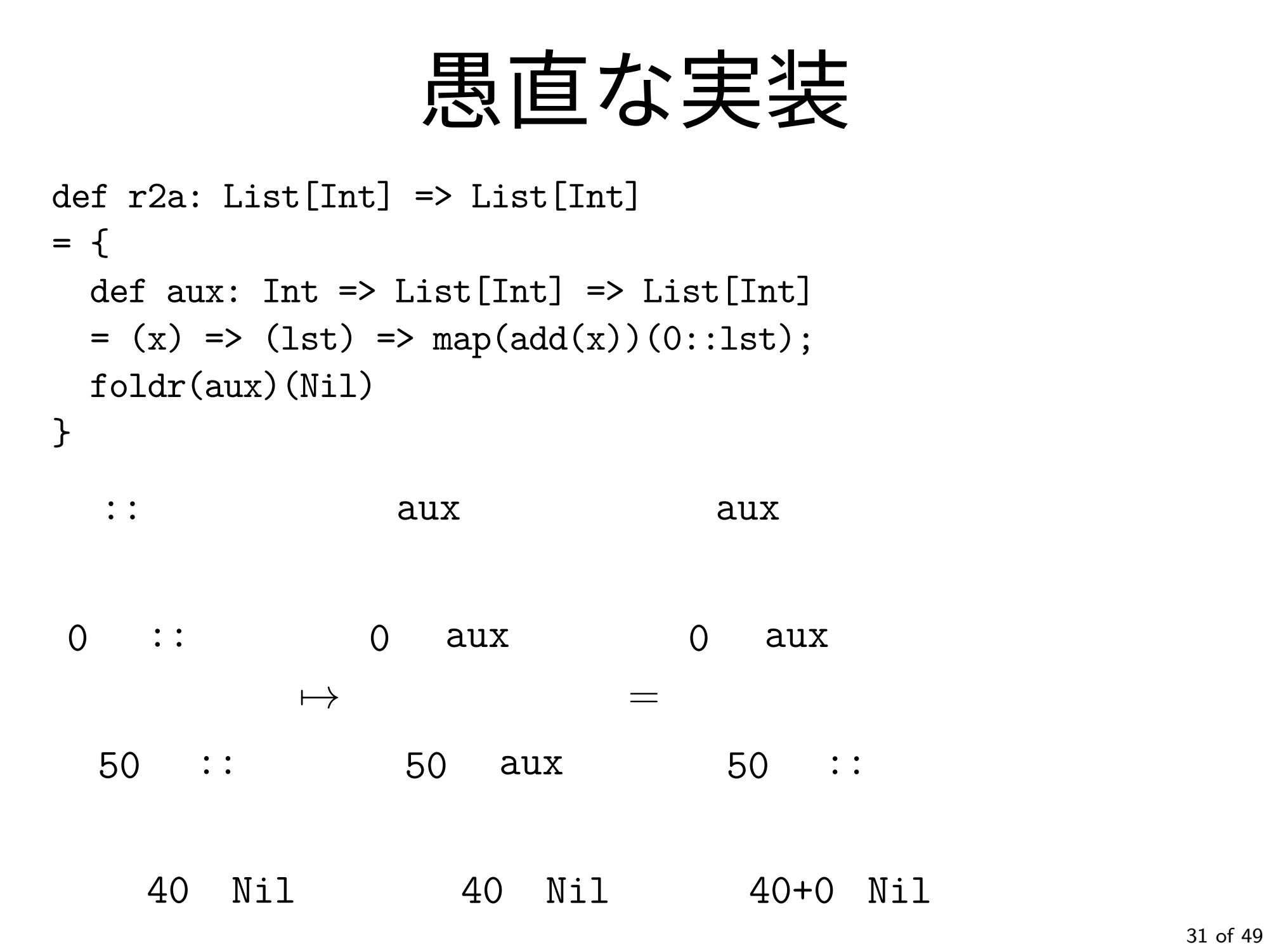
```
   ::              aux


 0   ::          0   aux
              ↦
  50   ::         50   aux


     40  Nil         40  Nil
```

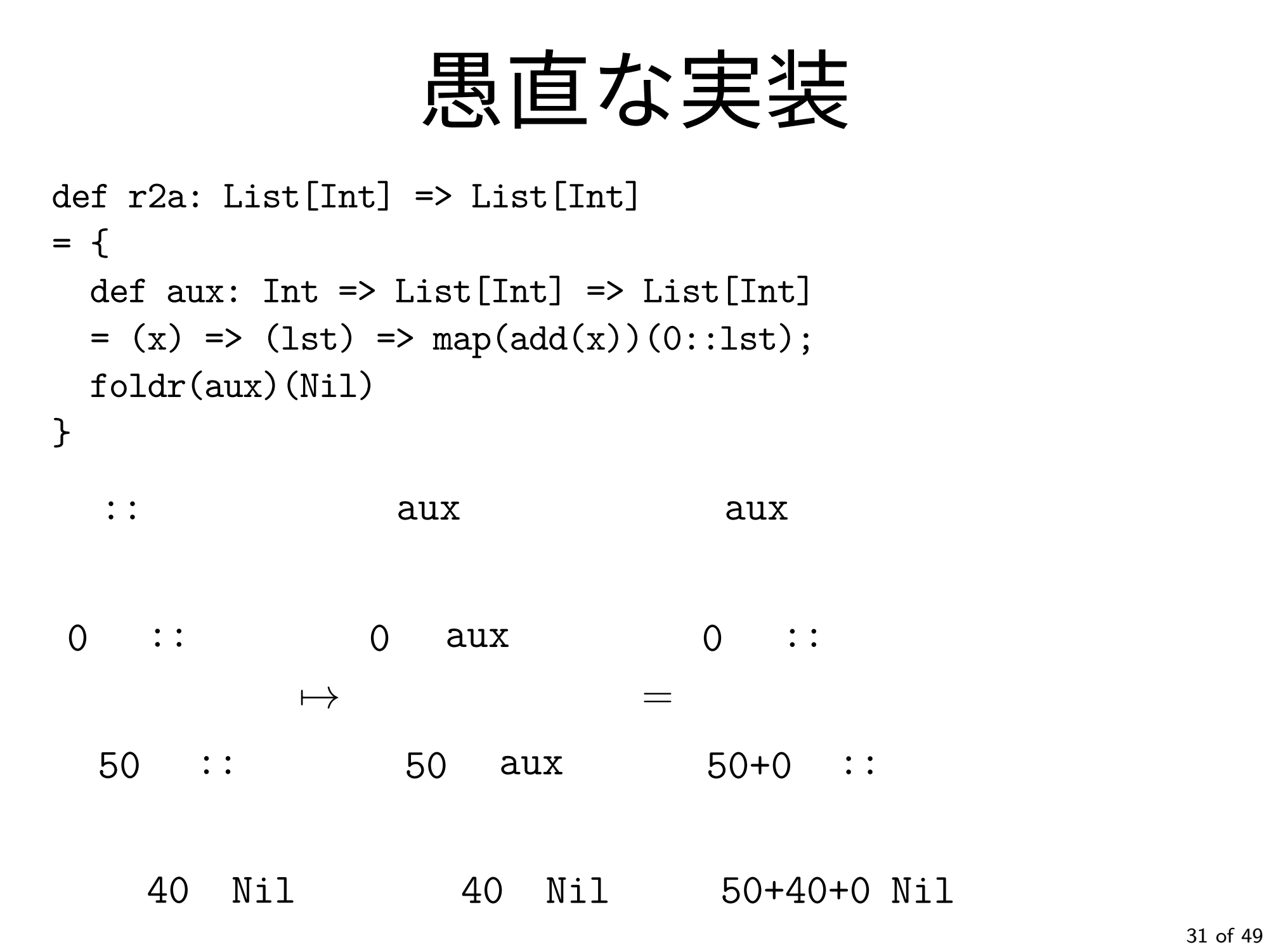
愚直な実装

```
def r2a: List[Int] => List[Int]
= {
  def aux: Int => List[Int] => List[Int]
  = (x) => (lst) => map(add(x))(0::lst);
  foldr(aux)(Nil)
}
```



愚直な実装

```
def r2a: List[Int] => List[Int]
= {
  def aux: Int => List[Int] => List[Int]
  = (x) => (lst) => map(add(x))(0::lst);
  foldr(aux)(Nil)
}
```



愚直な実装

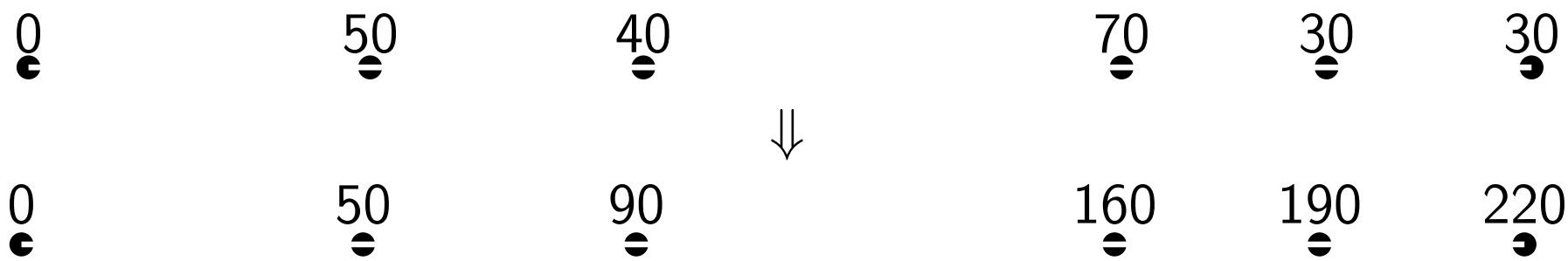
```
def r2a: List[Int] => List[Int]
= {
  def aux: Int => List[Int] => List[Int]
  = (x) => (lst) => map(add(x))(0::lst);
  foldr(aux)(Nil)
}
```

$$
\begin{array}{ccc}
:: & \text{aux} & :: \\
\\
0 :: & 0 \ \text{aux} & 0{+}0 :: \\
& \mapsto \quad = & \\
50 :: & 50 \ \text{aux} & 0{+}50{+}0 :: \\
\\
40 \ \text{Nil} & 40 \ \text{Nil} & 0{+}50{+}40{+}0 \ \text{Nil}
\end{array}
$$

愚直な実装の問題点: 非効率

加算の回数は点の数を nとして

- 手で計算: $O(n)$
- r2a: $O(n^2)$

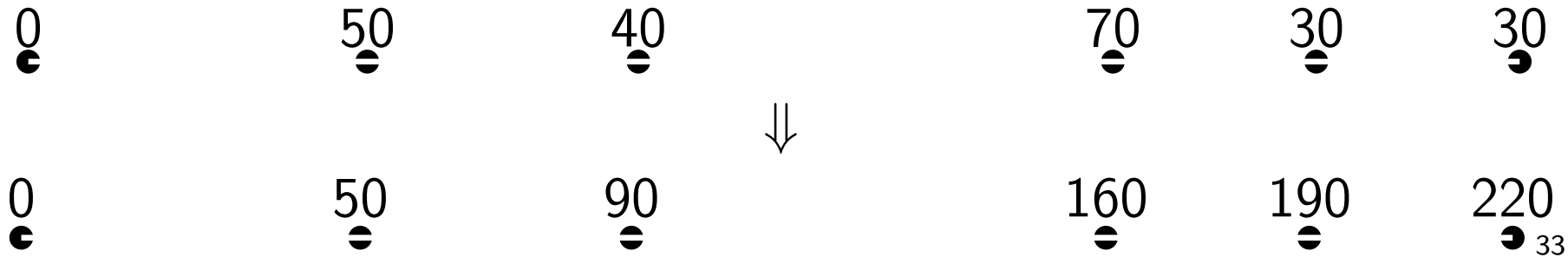



```
def r2a: List[Int] => List[Int]
= {
  def aux: Int => List[Int] => List[Int]
  = (x) => (lst) => map(add(x))(0::lst);
  foldr(aux)(Nil)
}
```

解決: 蓄積の利用

リストの左の要素から加算して計算結果を後続の計算に渡す
$\Rightarrow$ 加算の回数は $O(n)$

```
def r2a: List[Int] => List[Int] =
{
  def r2aAcc: Int => List[Int] =>  List[Int]
  = (acc) => (lst) => lst match{                    //accは蓄積
    case Nil => Nil
    case x::xs => (acc+x)::r2aAcc(acc+x)(xs)
  }
  r2aAcc(0)                                         //0は初期値
}
```



r2aAcc の振舞

```
def r2a: List[Int] => List[Int] =
{
  def r2aAcc: Int => List[Int] =>  List[Int]
  = (acc) => (lst) => lst match{                    //accは蓄積
    case Nil => Nil
    case x::xs => (acc+x)::r2aAcc(acc+x)(xs)
  }
  r2aAcc(0)                                         //0は初期値
}
```

```
 r2a(List(0,20,10))  =  r2aAcc(0)(List(0,20,10))
                     =  0::r2aAcc(0)(List(20,10))
                     =  0::20::r2aAcc(20)(List(10))
                     =  0::20::30::r2aAcc(30)(Nil)
                     =  0::20::30::Nil
```

非効率函数から効率的函数へ

非効率な函数の定義を分解して観察することで効率の良い函数を導くことが出来る (ただし直感・閃きが必要)

- `r2a(Nil)=Nil`
- `r2a(x::xs)`
  $=$ //`r2a` の定義を展開
  `foldr(aux)(Nil)(x::xs)`
  $=$ //`foldr(f)(n)(x::xs)`$\equiv$`f(x)(foldr(f)(n)(xs))`
  `aux(x)(foldr(aux)(Nil)(xs))`
  $=$ //`aux`$\triangleq$`(y)=>(lst)=>map(add(y))(0::lst)`
  `map(add(x))(0::foldr(aux)(Nil)(xs))`
  $=$ //`map(f)(x::xs)`$\equiv$`f(x)::map(f)(xs)`
  `x::map(add(x))(foldr(aux)(Nil)(xs))`
  $=$ //閃き (`aux` を展開して `foldr` と `map` の性質を利用)
  `x::foldr((y)=>(lst)=>map(add(y))(x::lst))(Nil)(xs)`

非効率函数から効率的函数へ

非効率な函数の定義を分解して観察することで効率の良い函数を導くことが出来る (ただし直感・閃きが必要)

- `r2a(Nil)=Nil`
- `r2a(x::xs)`
  `=`
  `foldr((y)=>(lst)=>map(add(y))(0::lst))(Nil)(x::xs)`
  `=`
  `x::foldr((y)=>(lst)=>map(add(y))(x::lst))(Nil)(xs)`

非効率函数から効率的函数へ

非効率な函数の定義を分解して観察することで効率の良い函数を導くことが出来る (ただし直感・閃きが必要)

- `r2a(Nil)=Nil`
- `r2a(x::xs)`
  `=`
  `foldr((y)=>(lst)=>map(add(y))(0::lst))(Nil)(x::xs)`
  `=`
  `x::foldr((y)=>(lst)=>map(add(y))(x::lst))(Nil)(xs)`

  $Q.$ `r2a(x::xs)=x::r2a(xs)` のような形にできないか？

非効率函数から効率的函数へ

非効率な函数の定義を分解して観察することで効率の良い函数を導くことが出来る (ただし直感・閃きが必要)

- `r2a(Nil)`$=$`Nil`
- `r2a(x::xs)`
  $=$
  `foldr((y)=>(lst)=>map(add(y))(0::lst))(Nil)(x::xs)`
  $=$
  `x::foldr((y)=>(lst)=>map(add(y))(x::lst))(Nil)(xs)`

  $Q.$ `r2a(x::xs)=x::r2a(xs)` のような形にできないか？
  $A.$ `0` と `x` の違いを吸収するための変数を一つ導入

非効率函数から効率的函数へ

非効率な函数の定義を分解して観察することで効率の良い函数を導くことが出来る (ただし直感・閃きが必要)

- `r2aAcc(acc)(Nil)=Nil`
- `r2aAcc(acc)(x::xs)`

  `=`

  `foldr((y)=>(lst)=>map(add(y))(acc::lst))(Nil)(x::xs)`

  `=`

  `...`

  `=`

  `(x+acc)::`
  `foldr((y)=>(lst)=>map(add(y))((x+acc)::lst))(Nil)(xs)`

  `=`

  `(x+acc)::r2aAcc(x+acc)(xs)`

蓄積を用いて効率化した函数

```
def r2a: List[Int] => List[Int] =
{
  def r2aAcc: Int => List[Int] =>  List[Int]
  = (acc) => (lst) => lst match{                    //accは蓄積
    case Nil => Nil
    case x::xs => (acc+x)::r2aAcc(acc+x)(xs)
  }
  r2aAcc(0)                                         //0は初期値
}
```

閃きの内容

`map(add(x))(foldr( (y)=>(l)=>map(add(y))(0::l) )(Nil)(ls))`

$\equiv$ `foldr( (y)=>(l)=>map(add(y))(add(x)(0)::l) )(Nil)(ls)`

正しさの証明
$\equiv$の左辺を `orig`，右辺を `opt` とする．`ls` の構造に関する帰納法で証明する．`aux`$\triangleq$`(y)=>(l)=>map(add(y))(0::l)` とする．

- `ls` $\equiv$ `Nil` のとき:
  `orig[Nil/ls]` $\equiv$ `Nil` $\equiv$ `opt[Nil/ls]`

閃きの内容

```
map(add(x))(foldr( (y)=>(l)=>map(add(y))(0::l) )(Nil)(ls))
≡ foldr( (y)=>(l)=>map(add(y))(add(x)(0)::l) )(Nil)(ls)
```

正しさの証明
≡の左辺を orig，右辺を opt とする．ls の構造に関する帰納法で証明する．aux$\triangleq$(y)=>(l)=>map(add(y))(0::l) とする．

- ls ≡ z::zs のとき (仮定は orig[zs/ls] ≡ opt[zs/ls]):

  ```
  orig
  ≡ map(add(x))(foldr(aux)(Nil)(z::zs))
  ≡ map(add(x))(aux(z)(foldr(aux)(Nil)(zs)))
  ≡ map(add(x))(map(add(z))(0::foldr(aux)(Nil)(zs)))
  ≡ map(add(z))(map(add(x))(0::foldr(aux)(Nil)(zs)))
  ≡ map(add(z))
    (add(x)(0)::map(add(x))(foldr(aux)(Nil)(zs)))
  ≡ map(add(z))(add(x)(0)::orig[zs/ls])
  ≡ map(add(z))(add(x)(0)::opt[zs/ls])
  ```

閃きの内容

```
map(add(x))(foldr( (y)=>(l)=>map(add(y))(0::l) )(Nil)(ls))
≡ foldr( (y)=>(l)=>map(add(y))(add(x)(0)::l) )(Nil)(ls)
```

正しさの証明
≡の左辺を orig，右辺を opt とする．ls の構造に関する帰納法で証明する．aux$\triangleq$(y)=>(l)=>map(add(y))(0::l) とする．

- ls ≡ z::zs のとき (仮定は orig[zs/ls] ≡ opt[zs/ls]):
  opt
  ≡ foldr((y)=>(l)=>map(add(y))(add(x)(0)::l))(Nil)
  (z::zs)
  ≡ map(add(z))(add(x)(0)::opt[zs/ls])
  ≡ orig □

例題 2: リストの反転

問題
リストが与えられた時，要素を逆順に並べたリストを返す函数 `reverse` を開発せよ

- `reverse(Nil)=Nil`
- `reverse(List(1,2,3))=List(3,2,1)`

例題 2: リストの反転

問題
リストが与えられた時，要素を逆順に並べたリストを返す函数 `reverse` を開発せよ

- `reverse(Nil)`$=$`Nil`
- `reverse(List(1,2,3))`$=$`List(3,2,1)`

愚直な実装 (`l++r`$\triangleq$`foldr(::)(r)(l)`)

```
def reverse[X]: List[X]=>List[X] = {
  def snoc: X=>List[X]=>List[X]
  = (x) => (lst) => {lst ++ List(x)} //xをlstの後ろにつける
  foldr(snoc)(Nil)
}
```

リストの長さを nとして，`::` の適用回数は $O(n^2) \Rightarrow$ 非効率

愚直な reverse の振舞

```
def reverse[X]: List[X]=>List[X] = {
  def snoc: X=>List[X]=>List[X]
  = (x) => (lst) => {lst ++ List(x)} //xをlstの後ろにつける
  foldr(snoc)(Nil)
}
```

```
    reverse(List(1,2,3))
 =  foldr(snoc)(Nil)(List(1,2,3))
 =  snoc(1)(foldr(snoc)(Nil)(List(2,3)))
 =  snoc(1)(snoc(2)(foldr(snoc)(Nil)(List(3))))
 =  snoc(1)(snoc(2)(snoc(3)(foldr(snoc)(Nil)(Nil))))
 =  snoc(1)(snoc(2)(snoc(3)(Nil)))
 =  snoc(1)(snoc(2)(Nil ++ List(3)))
 =  snoc(1)(snoc(2)(foldr(::)(List(3))(Nil)))
 =  snoc(1)(snoc(2)(List(3)))
 =  snoc(1)(foldr(::)(List(2))(List(3)))
 =  snoc(1)(3::foldr(::)(List(2))(Nil))
 =  snoc(1)(List(3,2))
```

愚直な reverse の振舞

```
def reverse[X]: List[X]=>List[X] = {
  def snoc: X=>List[X]=>List[X]
  = (x) => (lst) => {lst ++ List(x)} //xをlstの後ろにつける
  foldr(snoc)(Nil)
}
```

```
    reverse(List(1,2,3))
 =  ...
 =  snoc(1)(List(3,2))
 =  foldr(::)(List(1))(List(3,2))
 =  3::foldr(::)(List(1))(List(2))
 =  3::2::foldr(::)(List(1))(Nil)
 =  3::2::List(1)
 =  3::List(2,1)
 =  List(3,2,1)
```

蓄積を使った reverse

```
def reverse[X]: List[X]=>List[X] = {
  def revAcc: List[X]=>List[X]=>List[X]
  = (acc) => (lst) => lst match{      //accは反転したリスト
    case Nil => acc
    case x::xs => revAcc(x::acc)(xs)
  }
  revAcc(Nil)                          //蓄積の初期値はNil
}
```

リストの長さを nとして，:: の適用回数は $O(n)$

効率の良い reverse の導出

r2a と同じように効率の良い reverse を導くことができる

- reverse(Nil)=Nil
- reverse(x::xs)
  = //reverse の定義を展開，少し細工 (閃き)
  foldr(snoc)(Nil)(x::xs) ++ Nil
  = //foldr(f)(n)(x::xs)≡f(x)(foldr(f)(n)(xs))
  snoc(x)(foldr(snoc)(Nil)(xs)) ++ Nil
  = //snoc(x)(ls)≜ls ++ List(x)
  (foldr(snoc)(Nil)(xs) ++ List(x))++Nil
  = //(l1++l2)++l3=l1++(l2++l3) かつ l++Nil=l
  foldr(snoc)(Nil)(xs) ++ List(x)

効率の良い reverse の導出

r2a と同じように効率の良い reverse を導くことができる

- reverse(Nil)=Nil
- reverse(x::xs)
  = //reverse の定義を展開，少し細工 (閃き)
  foldr(snoc)(Nil)(x::xs) ++ Nil
  = //foldr(f)(n)(x::xs)≡f(x)(foldr(f)(n)(xs))
  snoc(x)(foldr(snoc)(Nil)(xs)) ++ Nil
  = //snoc(x)(ls)≜ls ++ List(x)
  (foldr(snoc)(Nil)(xs) ++ List(x))++Nil
  = //(l1++l2)++l3=l1++(l2++l3) かつ l++Nil=l
  foldr(snoc)(Nil)(xs) ++ List(x)

効率の良い reverse の導出

r2a と同じように効率の良い reverse を導くことができる

- reverse(Nil)=Nil
- reverse(x::xs)
  = //reverse の定義を展開，少し細工 (閃き)
  foldr(snoc)(Nil)(x::xs) ++ Nil
  = //foldr(f)(n)(x::xs)≡f(x)(foldr(f)(n)(xs))
  snoc(x)(foldr(snoc)(Nil)(xs)) ++ Nil
  = //snoc(x)(ls)≜ls ++ List(x)
  (foldr(snoc)(Nil)(xs) ++ List(x))++Nil
  = //(l1++l2)++l3=l1++(l2++l3) かつ l++Nil=l
  foldr(snoc)(Nil)(xs) ++ List(x)

  Nil と List(x) の違いを吸収する変数 acc を導入:
  revAcc(acc)(ls)≜foldr(snoc)(Nil)(ls) ++ acc
  ⇒ revAcc(acc)(x::xs)=revAcc(x::acc)(xs) を得る

revAcc の確認

revAcc(acc)(ls)≜foldr(snoc)(Nil)(ls)++acc の時，
revAcc(acc)(x::xs)=revAcc(x::acc)(xs) を確認する．

revAcc(acc)(x::xs)
= //revAcc の定義を展開
foldr(snoc)(acc)(x::xs)++acc
= //foldr(f)(n)(x::xs)≡f(x)(foldr(f)(n)(xs))
snoc(x)(foldr(snoc)(acc)(xs))++acc
= //snoc(x)(ls)≜ls ++ List(x)
(foldr(snoc)(acc)(xs)++List(x))++acc
= //(l1++l2)++l3=l1++(l2++l3)
foldr(snoc)(acc)(xs)++(List(x)++acc)
= //(List(a))++lst=a::lst
foldr(snoc)(acc)(xs)++(x::acc)
= //revAcc の定義
revAcc(x::acc)(xs)

蓄積を使った reverse

```
def reverse[X]: List[X]=>List[X] = {
  def revAcc: List[X]=>List[X]=>List[X]
  = (acc) => (lst) => lst match{      //accは反転したリスト
    case Nil => acc
    case x::xs => revAcc(x::acc)(xs)
  }
  revAcc(Nil)                          //蓄積の初期値はNil
}
```

リストの長さを nとして，:: の適用回数は $O(n)$

目次

部分構造を扱わない再帰函数
- 分割統治法 – クイックソートを例として
- 最大公約数の計算 – ユークリッドの互除法

バックトラックを用いたアルゴリズム

蓄積 (accumulation) の利用
- 閉路のあるグラフにおける経路探索
- 函数の効率化

課題

課題

以降の課題をとき，06.zip を OCW/OCW-i を使って提出せよ．
提出締め切りは5 月 26 日 15:00(JST)とする．

課題 1: maxprofit.scala

時点 t_0から t_nまでの株式会社 XYZ の株価がリストで与えられた時，株を高々一度ずつ売買して得られる最大の利潤を求めるプログラム `maxprofit` を開発せよ．ただし，0 以上の利潤が得られないときは売買を行わないものとし，最大の利潤を 0 とすること．株の購入時点を t_b，売却時点を t_sとしたとき，$b \leq s$でなければならないことに注意せよ (過去に戻って売却することはできない).

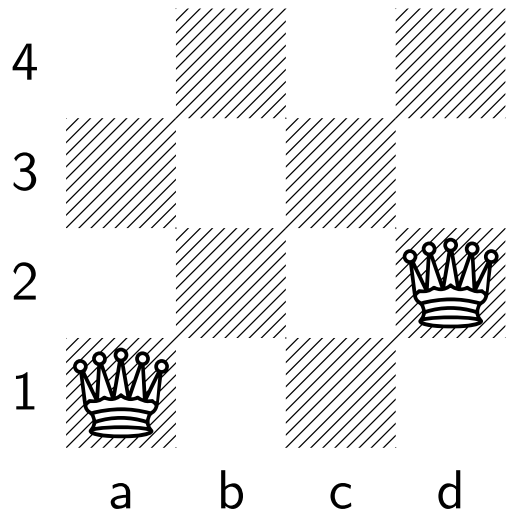
- `maxprofit(List(1,2,3,4,5)) == 4`
- `maxprofit(List(5,4,3,2,1)) == 0`
- `maxprofit(List(1,2,30,4,5)) == 29`
- `maxprofit(List(10,2,30,4,5)) == 28`
- `maxprofit(List(40,3,30,1,5)) == 27`

与えられるリストの長さ nに対して $O(n)$で計算するプログラムであることが望ましい (再帰函数の中で再帰函数を呼ぶと $O(n^2)$になる).

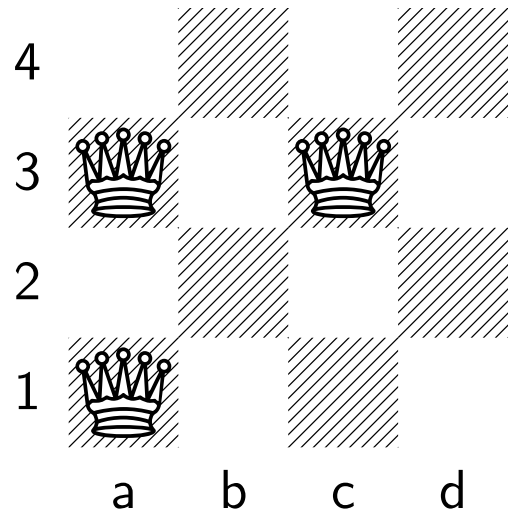
課題 2&3: 女王の配置

$n \times n$ のチェス盤に，m 個の女王を互いに取り合わないように配置するプログラムを開発する．位置 $(0,0)$ のマスを図の $(a,1)$ と解釈するものとする．女王は縦，横，斜め ($|$ 傾き $|=1$) 方向に任意マス進むことができる．

OK

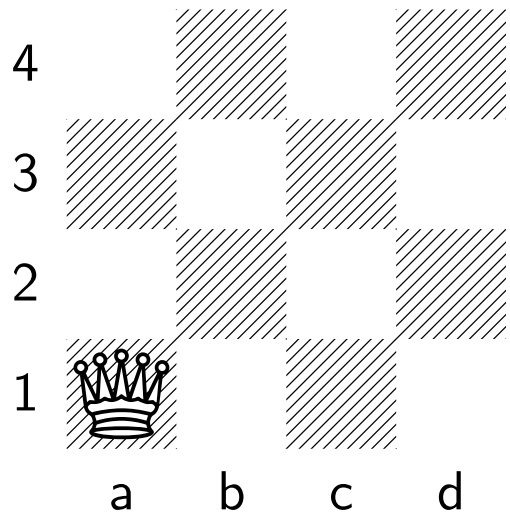


NG



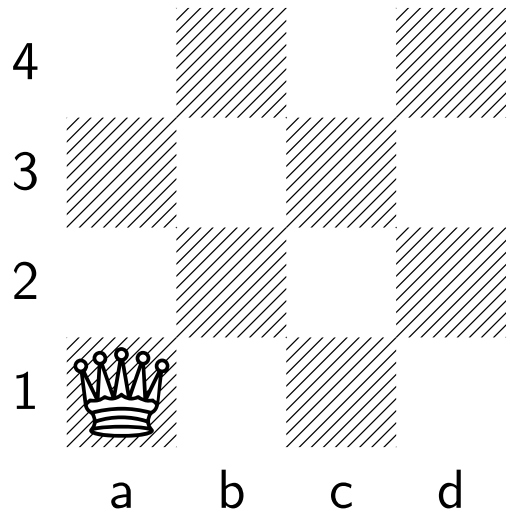
課題 2: board.scala

- チェス盤を表すデータ構造 `Board` を定義せよ
- 以下の函数を開発せよ.
  - `buildBoard: Int=>(Int=>Int=>Boolean)=>Board`
    チェス盤の縦 = 横のマス目の数 `n` と，女王の配置を示す函数 `queen` が与えられた時，`queen` に従って女王を配置した `n×n` のチェス盤を返す．`queen(x)(y)` が真の時，`(x,y)` に女王を配置する．偽の時は女王を配置しない．
    例: `buildBoard(4)(q)` with `q(x)(y)=true` if `x==0&&y==0`



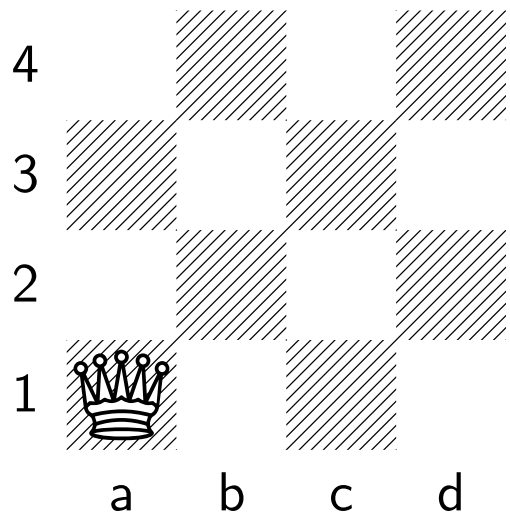
課題 2: board.scala

- チェス盤を表すデータ構造 `Board` を定義せよ
- 以下の函数を開発せよ.
  - `isThreatened: Board=>Int=>Int=>Boolean`
    チェス盤 `b` と整数値 `x`，`y` が与えられた時，チェス盤の `(x,y)` マスが，チェス盤の上にあるいずれかの女王の縦，横もしくは斜めにある場合は真，さもなくば偽を返す
    例: `isThreatend(b)(2)(2)==true` if `b==`:



課題 2: board.scala

- チェス盤を表すデータ構造 `Board` を定義せよ
- 以下の函数を開発せよ.
  - `isQueen: Board=>Int=>Int=>Boolean`
    チェス盤 `b` と 2 つの整数値 `x` および `y` が与えられた時，`b` の `(x,y)` に女王があれば真，なければ偽を返す.
    例: `isQueen(b)(0)(0)==true` if `b==`:



課題 3: placement.scala

自然数 n と女王の配置されたチェス盤 b が与えられた時，チェス盤 b に互いに取り合わないように n 個の女王が配置できる時，そのチェス盤を返し，さもなくば解なしと返すプログラム placement を開発せよ.

- placement の型: Int=>Board=>Option[Board], where

```
abstract class Option[A]
case object None extends Option[Nothing]
case class Some(a:A) extends Option[A]
```

  - 解があるときは解を Some でくるんだ値を返す
  - 解がないときは None を返す