計算機科学概論演習

#07

青谷知幸

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

May 26, 2016

目次

解説: 最大売却益の計算

蓄積 (accumulation) を利用した函数の作り方

数学的例題: 方程式の解を見つける
　　二分法

課題

目次

解説: 最大売却益の計算

蓄積 (accumulation) を利用した函数の作り方

数学的例題: 方程式の解を見つける
　　二分法

課題

最大売却益の計算

問題
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売却益を求めるプログラム `maxprofit` を開発せよ．ただし，0 以上の売却益が得られないときは売買を行わないものとし，最大の売却益を 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)．

最大売却益の計算

問題
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売却益を求めるプログラム `maxprofit` を開発せよ．ただし，0 以上の売却益が得られないときは売買を行わないものとし，最大の売却益を 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)．

愚直な解法
$\max\{p_b \mid 0 \leq b \leq n\}$ where
$p_b = \max\{price(t_s) - price(t_b) \mid b \leq s \leq n\}$

- p_bは時点 t_bで購入するときの最大売却益
- $price(t_i)$は時点 t_iの株価

愚直な解法の実装

数学的な定義

$\max\{p_b \mid 0 \leq b \leq n\}$ where

$p_b = \max\{price(t_s) - price(t_b) \mid b \leq s \leq n\}$

愚直な解法の実装

数学的な定義

$\max\{p_b \mid 0 \leq b \leq n\}$ where

$p_b = \max\{price(t_s) - price(t_b) \mid b \leq s \leq n\}$

p_bの計算

$$
\begin{aligned}
p_b &= \max\{price(t_s) - price(t_b) \mid b \leq s \leq n\} \\
&= \texttt{maxL(map(\_-}price(t_b)\texttt{)(List(}price(t_b)\texttt{,}\cdots\texttt{,}price(t_n)\texttt{)))} \\
&= \texttt{maxL(prof(}pl_b\texttt{))}
\end{aligned}
$$

ただし

$$
\begin{aligned}
\texttt{prof(x::xs)} &= \texttt{map(\_-x)(xs)} \\
\texttt{maxL} &= \texttt{foldr(max)(0)} \\
pl_i &= \texttt{List(}price(t_i)\texttt{,}\cdots\texttt{,}price(t_n)\texttt{)}
\end{aligned}
$$

愚直な解法の実装

数学的な定義

$\max\{p_b \mid 0 \leq b \leq n\}$ where

$p_b = \max\{price(t_s) - price(t_b) \mid b \leq s \leq n\}$

p_bの計算

$$
\begin{aligned}
p_b &= \max\{price(t_s) - price(t_b) \mid b \leq s \leq n\} \\
&= \texttt{maxL(map(\_-}price(t_b)\texttt{)(List(}price(t_b)\texttt{,}\cdots\texttt{,}price(t_n)\texttt{)))} \\
&= \texttt{maxL(prof(}pl_b\texttt{))}
\end{aligned}
$$

ただし

$$
\begin{aligned}
\texttt{prof(x::xs)} &= \texttt{map(\_-x)(xs)} \\
\texttt{maxL} &= \texttt{foldr(max)(0)} \\
pl_i &= \texttt{List(}price(t_i)\texttt{,}\cdots\texttt{,}price(t_n)\texttt{)}
\end{aligned}
$$

maxprofit の計算

$$
\begin{aligned}
\max\{p_b \mid 0 \leq b \leq n\} &= \texttt{maxL(List(}p_0\texttt{,}\cdots\texttt{,}p_n\texttt{))} \\
&= \texttt{maxL(map(prof)(List(}pl_0\texttt{,}\cdots\texttt{,}pl_n\texttt{)))}
\end{aligned}
$$

愚直な解法の実装

```
def tails[X]: List[X]=>List[List[X]] //List(pl_0,...,pl_n)を作る
= (lst) => lst match{
  case Nil => List(Nil)
  case x::xs => lst::tails(xs)
}//tails(List(1,2,3))==List(List(1,2,3),List(2,3),List(3),Nil)

def maxprofit: List[Int] => Int
= (lst) => {
  def aux: List[Int] => Int = (lst) => lst match{ //p_bの計算
    case Nil => 0
    case x::xs => foldr(max)(0)(xs) - x
  }
  foldr(max)(0)(map(aux)(tails(lst)))
}
```

愚直な解法の実装

```
def tails[X]: List[X]=>List[List[X]] //List(pl_0,···,pl_n) を作る
= (lst) => lst match{
  case Nil => List(Nil)
  case x::xs => lst::tails(xs)
}//tails(List(1,2,3))==List(List(1,2,3),List(2,3),List(3),Nil)

def maxprofit: List[Int] => Int
= (lst) => {
  def aux: List[Int] => Int = (lst) => lst match{ //p_bの計算
    case Nil => 0
    case x::xs => foldr(max)(0)(xs) - x
  }
  foldr(max)(0)(map(aux)(tails(lst)))
}
```

計算量: 入力リストの長さ nに対して $O(n^2)$

$O(n)$のプログラム

```
def maxprofit: List[Int] => Int
= (lst) => lst match{
  case Nil => 0
  case x::Nil => 0
  case x::y::xs => max(y-x)(maxprofit((min(x)(y))::xs))
}
```

- リストの先頭要素 (x) は株価の最安値 (蓄積)
- maxprofit を呼び出す度にリストが 1 ずつ短くなる
  $\Rightarrow$ 高々 n回の再帰で止まる
- maxprofit 一度につき減算が 1 回，min が 1 回, max が 1 回
  $\Rightarrow$ maxprofit 以外に再帰函数はなし

$O(n^2)$から $O(n)$へ

この問題においても $O(n^2)$のプログラムから $O(n)$のプログラムを導き出すことができる．入力のリストを以下の場合についてそれぞれ考察する．

- `Nil`
- `x::Nil`
- `x::y::xs`

```
def maxprofit: List[Int] => Int
= (lst) => {
  def aux: List[Int] => Int = (lst) => lst match{ //pbの計算
    case Nil => 0
    case x::xs => foldr(max)(0)(xs)-x
  }
  foldr(max)(0)(map(aux)(tails(lst)))
}
```

$O(n^2)$から $O(n)$へ

```
def maxprofit: List[Int] => Int
= (lst) => {
  def aux: List[Int] => Int = (lst) => lst match{ //p_bの計算
    case Nil => 0
    case x::xs => foldr(max)(0)(xs)-x
  }
  foldr(max)(0)(map(aux)(tails(lst)))
}
```

与えられたリストが Nil の時:

maxprofit(Nil) $=$ 0 (tails(Nil)==List(Nil))

$O(n^2)$から $O(n)$へ

```
def maxprofit: List[Int] => Int
= (lst) => {
  def aux: List[Int] => Int = (lst) => lst match{ //p_bの計算
    case Nil => 0
    case x::xs => foldr(max)(0)(xs)-x
  }
  foldr(max)(0)(map(aux)(tails(lst)))
}
```

与えられたリストが `x::Nil` に分解できる時:

`maxprofit(x::Nil)` $=$ `0`

`(tails(x::Nil)==List(List(x),Nil)==List(x)::Nil::Nil)`

$O(n^2)$から $O(n)$へ

与えられたリストが x::y::xs に分解できる時:

```
maxprofit(x::y::xs)
= //maxprofit(lst)≜foldr(max)(0)(map(aux)(tails(lst)))
foldr(max)(0)(map(aux)(tails(x::y::xs)))
= //tails(x::xs)≜(x::xs)::tails(xs)
foldr(max)(0)(map(aux)((x::y::xs)::tails(y::xs)))
= //map(f)(x::xs)≜f(x)::map(f)(xs)
foldr(max)(0)(aux(x::y::xs)::map(aux)(tails(y::xs)))
= //foldr(f)(n)(x::xs)≜f(x)(foldr(f)(n)(xs))
max(aux(x::y::xs))(foldr(max)(0)(map(aux)(tails(y::xs))))
= //tails(x::xs)≜(x::xs)::tails(xs)
max(aux(x::y::xs))
    (foldr(max)(0)(map(aux)((y::xs)::tails(xs))))
= //map(f)(x::xs)≜f(x)::map(f)(xs)
max(aux(x::y::xs))
    (foldr(max)(0)(aux(y::xs)::map(aux)(tails(xs))))
```

$O(n^2)$から $O(n)$へ

```
maxprofit(x::y::xs)
= //...
max(aux(x::y::xs))
    (foldr(max)(0)(aux(y::xs)::map(aux)(tails(xs))))
= //foldr(f)(n)(x::xs)≜f(x)(foldr(f)(n)(xs))
max(aux(x::y::xs))
    (max(aux(y::xs))(foldr(max)(0)(map(aux)(tails(xs)))))
= //aux(x::xs)≜foldr(max)(0)(xs)-x
max(foldr(max)(0)(y::xs)-x)
    (max(foldr(max)(0)(xs)-y)
        (foldr(max)(0)(map(aux)(tails(xs)))))
= //foldr(f)(n)(x::xs)≜f(x)(foldr(f)(n)(xs))
max(max(y)(foldr(max)(0)(xs))-x)
    (max(foldr(max)(0)(xs)-y)
        (foldr(max)(0)(map(aux)(tails(xs)))))
```

$O(n^2)$から $O(n)$へ

```
maxprofit(x::y::xs)
= //...
max(max(y)(foldr(max)(0)(xs))-x)
    (max(foldr(max)(0)(xs)-y)
        (foldr(max)(0)(map(aux)(tails(xs)))))
= //max(x)(y)-z≡max(x-z)(y-z)
max(max(y-x)(foldr(max)(0)(xs)-x))
    (max(foldr(max)(0)(xs)-y)
        (foldr(max)(0)(map(aux)(tails(xs)))))
= //max(max(x)(y))(z)≡max(x)(max(y)(z))
max(max(y-x)
        (max(foldr(max)(0)(xs)-x)(foldr(max)(0)(xs)-y))
    (foldr(max)(0)(map(aux)(tails(xs)))))
= //max(z-x)(z-y)≡z-min(x)(y)
max(max(y-x)(foldr(max)(0)(xs)-min(x)(y))
    (foldr(max)(0)(map(aux)(tails(xs)))))
```

$O(n^2)$から $O(n)$へ

```
maxprofit(x::y::xs)
= //...
max(max(y-x)(foldr(max)(0)(xs)-min(x)(y))
    (foldr(max)(0)(map(aux)(tails(xs)))))
= //aux(x::xs)≜foldr(max)(0)(xs)-x
max(max(y-x)(aux(min(x)(y)::xs))
    (foldr(max)(0)(map(aux)(tails(xs)))))
= //max(max(x)(y))(z)≡max(x)(max(y)(z))
max(y-x)
    (max(aux(min(x)(y)::xs))
        (foldr(max)(0)(map(aux)(tails(xs)))))
= //foldr(f)(n)(x::xs)≜f(x)(foldr(f)(n)(xs))
max(y-x)
    (foldr(max)(0)
        (aux(min(x)(y)::xs)::map(aux)(tails(xs))))
```

$O(n^2)$から $O(n)$へ

```
maxprofit(x::y::xs)
= //...
max(y-x)
    (foldr(max)(0)
        (aux(min(x)(y)::xs)::map(aux)(tails(xs))))
= //map(f)(x::xs)≜f(x)::map(f)(xs)
max(y-x)
    (foldr(max)(0)
        (map(aux)((min(x)(y)::xs)::tails(xs))))
= //tails(x::xs)≜(x::xs)::tails(xs)
max(y-x)
    (foldr(max)(0)(map(aux)(tails(min(x)(y)::xs))))
= //maxprofit(lst)≜foldr(max)(0)(map(aux)(tails(lst)))
max(y-x)(maxprofit(min(x)(y)::xs))
```

目次

解説: 最大売却益の計算

蓄積 (accumulation) を利用した函数の作り方

数学的例題: 方程式の解を見つける
　　二分法

課題

蓄積の必要を見抜く

蓄積が必要となる状況は多様である．典型的な状況例は:

- 部分構造を扱う再帰函数 (structural recursion):
  再帰函数の中で別の再帰函数を呼ぶ

```
def snoc: (X,List[X])=>List[X] = (e,lst) => lst match{
  case Nil => List(e)
  case x::xs => x::snoc(e,xs)
}
def reverse[X]: List[X]=>List[X] = (lst) => lst match{
  case Nil => Nil
  case x::xs => snoc(x,reverse(xs))
}
```

- 部分構造を扱わない再帰函数 (generative recursion):
  出力が得られるはずの入力に対して出力が得られない (無限ループ)

蓄積を利用した函数の作り方

プログラム導出に依らずに蓄積を利用した函数を作るには，

1. 蓄積データの決定
2. 蓄積を利用した計算方法の決定

以降では reverse を例題にこの方法を説明する

```
def snoc: (X,List[X])=>List[X] = (e,lst) => lst match{
  case Nil => List(e)
  case x::xs => x::snoc(e,xs)
}
def reverse[X]: List[X]=>List[X] = (lst) => lst match{
  case Nil => Nil
  case x::xs => snoc(x,reverse(xs))
}
```

蓄積データの決定

以下の二つを明らかにする

- 蓄える情報/知識
- 情報/知識の蓄え方

この 2 つを明らかにするために，函数内函数として蓄積を利用する函数の雛型を用意する．この函数の引数は元の函数の引数に蓄積を加えたものである:

```
def reverse[X]: List[X] => List[X]
= (lst) => {
  def rev: List[X] => Accumulator => //Accumulatorは蓄積の型
  = (lst0) => (acc) => lst0 match{
    case Nil => ...
    case x::xs => ... rev(xs)(... x ... acc ...) ...
  }
  rev(lst)(...) //蓄積の初期値は？
}
```

蓄える情報と蓄え方の決定

```
def reverse[X]: List[X] => List[X]
= (lst) => {
  def rev: List[X] => Accumulator => //Accumulatorは蓄積の型
  = (lst0) => (acc) => lst0 match{
    case Nil => ...
    case x::xs => ... rev(xs)(... x ... acc ...) ...
  }
  rev(lst)(...) //蓄積の初期値は？
}
```

`rev` は与えられたリストを反転させるから，蓄積はこれまでに処理した要素が全て蓄えられていなければならない．従って，

- 蓄える情報: `lst` の要素で `lst0` より前にあるもの全て
- 蓄え方: `List[X]`, i.e., `type Accumulator = List[X]`

蓄える情報と蓄え方の決定

```
def reverse[X]: List[X] => List[X]
= (lst) => {
  type Accumulator = List[X]
  def rev: List[X] => Accumulator => //Accumulatorは蓄積の型
  = (lst0) => (acc) => lst0 match{
    case Nil => ...
    case x::xs => ... rev(xs)(x::acc) ...
  }
  rev(lst)(...) //蓄積の初期値は？
}
```

rev は与えられたリストを反転させるから，蓄積はこれまでに処理した要素が全て蓄えられていなければならない．従って，

- 蓄える情報: lst の要素で lst0 より前にあるもの全て
- 蓄え方: List[X], i.e., type Accumulator = List[X]

蓄積の初期値の決定

蓄える情報は “lst の要素で lst0 より前にあるもの全て” だから，初期値は空のリストになる．

```
def reverse[X]: List[X] => List[X]
= (lst) => {
  type Accumulator = List[X]
  def rev: List[X] => Accumulator => //Accumulatorは蓄積の型
  = (lst0) => (acc) => lst0 match{
    case Nil => ...
    case x::xs => ... rev(xs)(x::acc) ...
  }
  rev(lst)(Nil)
}
```

蓄積利用の計算方法の決定

蓄える情報は “lst の要素で lst0 より前にあるもの全て” ⇒

- lst0==Nil ⇒ acc をそのまま返す
- lst0!=Nil ⇒ rev の計算結果をそのまま返す

```
def reverse[X]: List[X] => List[X]
= (lst) => {
  type Accumulator = List[X]
  def rev: List[X] => Accumulator => //Accumulatorは蓄積の型
  = (lst0) => (acc) => lst0 match{
    case Nil => ...
    case x::xs => ... rev(xs)(x::acc) ...
  }
  rev(lst)(Nil)
}
```

蓄積利用の計算方法の決定

蓄える情報は “lst の要素で lst0 より前にあるもの全て” ⇒

- lst0==Nil ⇒ acc をそのまま返す
- lst0!=Nil ⇒ rev の計算結果をそのまま返す

```
def reverse[X]: List[X] => List[X]
= (lst) => {
  type Accumulator = List[X]
  def rev: List[X] => Accumulator => //Accumulatorは蓄積の型
  = (lst0) => (acc) => lst0 match{
    case Nil => acc
    case x::xs => rev(xs)(x::acc)
  }
  rev(lst)(Nil)
}
```

例題: n進数から 10 進数へ

問題
n進数表記の数がリスト (digits と呼ぶことにする) で与えられる．digits の長さを lとして，i番目の値は $l-1-i$桁目の値を表すものとする．digits を 10 進数表記の整数値 (Int) に変換する函数 to10 を開発せよ．

- `to10(10)(List(1,0,2))==102`
- `to10(8)(List(1,0,2))==66`

例題: n進数から 10 進数へ

問題
n進数表記の数がリスト (digits と呼ぶことにする) で与えられる．digits の長さを lとして，i番目の値は $l-1-i$桁目の値を表すものとする．digits を 10 進数表記の整数値 (Int) に変換する函数 to10 を開発せよ．

- to10(10)(List(1,0,2))==102
- to10(8)(List(1,0,2))==66

数学的な定義

$$\texttt{to10(n)(digits)} \triangleq \sum_{i=0}^{|\texttt{digits}|-1} n^i \cdot \texttt{nth}(|\texttt{digits}| - 1 - \texttt{i}, \texttt{digits})$$

ただし

$$\texttt{nth}(\texttt{i}, \texttt{digits}) \triangleq \begin{cases} \texttt{head}(\texttt{digits}) & \text{if } i = 0 \\ \texttt{nth}(\texttt{i} - 1, \texttt{tail}(\texttt{digits})) & \text{otherwise} \end{cases}$$

to10: 愚直な実装

数学的な定義

$$\texttt{to10(n)(digits)} \triangleq \sum_{i=0}^{|\texttt{digits}|-1} n^i \cdot \texttt{nth}(|\texttt{digits}| - 1 - \texttt{i}, \texttt{digits})$$

ただし

$$\texttt{nth}(\texttt{i}, \texttt{digits}) \triangleq \begin{cases} \texttt{head}(\texttt{digits}) & \text{if } i = 0 \\ \texttt{nth}(\texttt{i} - 1, \texttt{tail}(\texttt{digits})) & \text{otherwise} \end{cases}$$

プログラム

```
def nth[X]: Int=>List[X]=>X = (n)=>(digits)=>digits match{
  case x::xs => if(n==0) x else nth(n-1)(xs)
}
def to10: Int=>List[Int]=>Int = (n)=>(digits)=>{
  val len=length(digits)
  def aux: Int=>Int = (i)=>{pow(n)(i)*nth(len-1-i)(digits)}
  foldr(add)(0)(map(aux)(range(succ)(0)(len)))
}
```

to10: 愚直な実装

数学的な定義

$$\texttt{to10(n)(digits)} \triangleq \sum_{i=0}^{|\texttt{digits}|-1} n^i \cdot \texttt{nth}(|\texttt{digits}| - 1 - \texttt{i}, \texttt{digits})$$

ただし

$$\texttt{nth}(\texttt{i}, \texttt{digits}) \triangleq \begin{cases} \texttt{head}(\texttt{digits}) & \text{if } i = 0 \\ \texttt{nth}(\texttt{i} - 1, \texttt{tail}(\texttt{digits})) & \text{otherwise} \end{cases}$$

プログラム

```
def nth[X]: Int=>List[X]=>X = (n)=>(digits)=>digits match{
  case x::xs => if(n==0) x else nth(n-1)(xs)
}
def to10: Int=>List[Int]=>Int = (n)=>(digits)=>{
  val len=length(digits)
  def aux: Int=>Int = (i)=>{pow(n)(i)*nth(len-1-i)(digits)}
  foldr(add)(0)(map(aux)(range(succ)(0)(len)))
}
```

map の中で再帰函数 aux を呼んでいる ⇒ 累積を使う

to10: 雛型の導入

蓄積を利用した函数の雛型を導入する．*n*進数の *n*は再帰呼出しの間で変わりようがないため，*n*以外の引数と蓄積を受け取る雛型を書く:

```
def to10: Int=>List[Int]=>Int
= (n) => (digits) => {
  def to10Acc: List[Int]=>Accumulator=>Int
  = (digits0) => (acc) => digits match{
    case Nil => ...
    case x::xs => ... to10Acc(xs)(... x ... acc ...) ...
  }
  to10Acc(digits)(...)
}
```

蓄積する情報と方法の決定

```
def to10: Int=>List[Int]=>Int
= (n) => (digits) => {
  def to10Acc: List[Int]=>Accumulator=>Int
  = (digits0) => (acc) => digits match{
    case Nil => ...
    case x::xs => ... to10Acc(xs)(... x ... acc ...) ...
  }
  to10Acc(digits)(...)
}
```

- 蓄積する情報:
  digits≡prefix++digits0 となる prefix の Int 値，つまり acc≡to10(n)(prefix)
- 蓄積の方法: Int

蓄積する情報と方法の決定

```
def to10: Int=>List[Int]=>Int
= (n) => (digits) => {
  type Accumulator = Int
  def to10Acc: List[Int]=>Accumulator=>Int
  = (digits0) => (acc) => digits match{
    case Nil => ...
    case x::xs => ... to10Acc(xs)(x+acc*n) ...
  }
  to10Acc(digits)(...)
}
```

- 蓄積する情報:
  digits≡prefix++digits0 となる prefix の Int 値，つまり acc≡to10(n)(prefix)
- 蓄積の方法: Int

蓄積の初期値

```
def to10: Int=>List[Int]=>Int
= (n) => (digits) => {
  type Accumulator = Int
  def to10Acc: List[Int]=>Accumulator=>Int
  = (digits0) => (acc) => digits match{
    case Nil => ...
    case x::xs => ... to10Acc(xs)(x+acc*n) ...
  }
  to10Acc(digits)(...)
}
```

acc≡to10(n)(prefix) より初期値は to10(n)(Nil)=0

蓄積の初期値

```
def to10: Int=>List[Int]=>Int
= (n) => (digits) => {
  type Accumulator = Int
  def to10Acc: List[Int]=>Accumulator=>Int
  = (digits0) => (acc) => digits match{
    case Nil => ...
    case x::xs => ... to10Acc(xs)(x+acc*n) ...
  }
  to10Acc(digits)(0)
}
```

acc≡to10(n)(prefix) より初期値は to10(n)(Nil)=0

蓄積利用の計算方法の決定

```
def to10: Int=>List[Int]=>Int
= (n) => (digits) => {
  type Accumulator = Int
  def to10Acc: List[Int]=>Accumulator=>Int
  = (digits0) => (acc) => digits match{
    case Nil => ...
    case x::xs => ... to10Acc(xs)(x+acc*n) ...
  }
  to10Acc(digits)(0)
}
```

- digits0==Nil: acc を返す
- digits0!=Nil: to10Acc の結果をそのまま返す

蓄積利用の計算方法の決定

```
def to10: Int=>List[Int]=>Int
= (n) => (digits) => {
  type Accumulator = Int
  def to10Acc: List[Int]=>Accumulator=>Int
  = (digits0) => (acc) => digits match{
    case Nil => acc
    case x::xs => to10Acc(xs)(x+acc*n)
  }
  to10Acc(digits)(0)
}
```

- digits0==Nil: acc を返す
- digits0!=Nil: to10Acc の結果をそのまま返す

目次

解説: 最大売却益の計算

蓄積 (accumulation) を利用した函数の作り方

数学的例題: 方程式の解を見つける
　　二分法

課題

二分法

函数 $f(x)$を連続函数．$a < b$について，$f(a)$と $f(b)$の符号が異なる時，$f(r) = 0$となる根 rが区間 $[a, b]$に存在する．二分法はこの事実を用いて，以下の性質を用いて根の存在範囲を繰り返し狭め，根を求める．

根の性質

aと bの中間値を mとする．$f(a)$と $f(m)$の符号が異なる時，根は区間 $[a, m]$にある．さもなくば $[m, b]$にある．

二分法の実現: `findRoot`

根の性質

aと bの中間値を mとする．$f(a)$と $f(m)$の符号が異なる時，根は区間 $[a, m]$にある．さもなくば $[m, b]$にある．

コード

```
def findRoot: (Double => Double) => (Double, Double) => Double
= (f) => (a,b) =>
/**
 * 目的: 函数f(x)==0となるxを[a,b]を狭めることで求める
 */
{...}
```

二分法の実現: findRoot

根の性質

aと bの中間値を mとする．$f(a)$と $f(m)$の符号が異なる時，根は区間 $[a, m]$にある．さもなくば $[m, b]$にある．

コード

```
def findRoot: (Double => Double, Double, Double) => Double
= (f, a, b) =>
/**
 * 目的: 函数f(x)==0となるxを[a,b]を狭めることで求める
 * テンプレート:
 *     if(自明に解が求まる) ...
 *     else if( f(m)*f(a) < 0 ) findRoot(...)
 *     else findRoot(...)
 */
{...}
```

Q. 自明に解が求まるのはいつ?

二分法の実現: findRoot

根の性質

aと bの中間値を mとする．$f(a)$と $f(m)$の符号が異なる時，根は区間 $[a, m]$にある．さもなくば $[m, b]$にある．

コード

```
def findRoot: (Double => Double, Double, Double) => Double
= (f, a, b) =>
/**
 * 目的: 函数f(x)==0となるxを[a,b]を狭めることで求める
 * テンプレート:
 *     if(自明に解が求まる) ...
 *     else if( f(m)*f(a) < 0 ) findRoot(...)
 *     else findRoot(...)
 */
{...}
```

$Q.$ 自明に解が求まるのはいつ? $A.$ 区間が十分に狭い時

二分法の実現: findRoot

根の性質

aと bの中間値を mとする．$f(a)$と $f(m)$の符号が異なる時，根は区間 $[a, m]$にある．さもなくば $[m, b]$にある．

コード

```
def TOLERANCE : Double = 0.00001
def findRoot: (Double => Double, Double, Double) => Double
= (f, a, b) =>
/**
 * 目的: 函数f(x)==0となるxを[a,b]を狭めることで求める
 * テンプレート:
 *     if(b - a <= TOLERANCE) ...
 *     else if( f(m)*f(a) < 0 ) findRoot(...)
 *     else findRoot(...)
 */
{...}
```

二分法の実現: findRoot

根の性質

aと bの中間値を mとする．$f(a)$と $f(m)$の符号が異なる時，根は区間 $[a, m]$にある．さもなくば $[m, b]$にある．

コード

```
def TOLERANCE = 0.0001
def findRoot: (Double => Double, Double, Double) => Double
= (f, a, b) => {
  if(b - a <= TOLERANCE) a
  else{
    def m = (a+b)*0.5                    //局所的に中間値を定義
    if(f(m) * f(a) < 0) findRoot(f, a, m)
    else findRoot(f, m, b)
  }}
```

二分法の注意点

二分法は根を近似的に求めるだけである

例: $f(x) = x^3 - 27$

```
scala> def cube = (x:Double) => {Math.pow(x,3)-27}
cube: Double => Double

scala> findRoot(cube,0,5)
res0: Double = 2.9999542236328125
```

目次

解説: 最大売却益の計算

蓄積 (accumulation) を利用した函数の作り方

数学的例題: 方程式の解を見つける
　　二分法

課題

課題

以降の課題をとき，07.zip を OCW/OCW-i を使って提出せよ．
提出締め切りは6 月 1 日 15:00(JST)とする．

課題 1: 人喰いと探検家 (guide.scala)

問題
宝の地図を携えた n人の探検家が n人の人喰いに案内されて森を歩いている．宝の地点まで後もう一歩というところで，彼らは危険な蛇と魚に満ちた幅の広い河に行きあたった．河を渡る以外に進む方法はなく，河を渡るためには近場で見つけた m人乗りの舟を使うしかない．岸から岸へ舟を渡すためには舟を漕ぐしかなく，岸の両側において，人喰いの人数が探検家よりも多くなると，人喰いは直ちに探検家を殺めて食べてしまう．$2n$人全員が対岸へ渡る方法を提示するプログラム `solve` を開発し，探検家を救ってください．舟を移動させるためには少なくとも一人乗船していなければならず，舟で対岸へ移動したら，必ず一度全員降りるものとする．また，舟の上では探検家は喰われない．

手順

1. それぞれの岸にいる探検家と人喰いの数，舟が運ぶことのできる最大の人数，そしてどちらの岸に舟があるかを表現するデータ構造 `State` を定義せよ
2. 探検家と人喰いの数，舟の位置と舟に乗ることのできる最大の人数がそれぞれ与えられた時，その状況を表現する `State` 値を作る函数 `makeState` を定義せよ．
3. 片岸から片岸へ 1 度舟を動かすとき，舟で運ぶ探検家と人喰いの人数を表現するデータ構造 `Load` を定義せよ
4. `State` を受け取り，一度舟を対岸へ動かして探検家と人喰いを移動させることで生み出される可能な `State` とその運搬方法 `Load` を列挙する函数 `next` を定義せよ．
5. `State` を受け取り，それが表す状況が正常 (だれも喰われない) かどうか判定する函数 `isLeagal` を定義せよ

手順

6. State を受け取り，全員を渡し終えたかどうかを判定する函数 isFinal を定義せよ
7. 探検家と人喰いの人数および舟の大きさ (最大人数) を受け取り，渡し方が存在する場合にはその渡し方を表すリストを Some で包んで，渡し方が存在しない場合には None を返す函数 solve を定義せよ．solve の型は Int=>Int=>Option[List[Load]] であり，第一引数は探検家と人喰いの人数，第二引数は舟が運ぶことのできる最大の人数を表す．探検家と人喰いと舟は全て同じ岸にあるものとせよ．返値は
   - 全員を対岸へ運ぶ方法がないとき None
   - 全員を対岸へ運ぶ方法があるときその方法を Some で包んだもの，つまり Some(List($\mathtt{Load}_1$,$\cdots$,$\mathtt{Load}_n$))

課題 2: triangular.scala

連立方程式の係数と定数から作った行列が
`List[List[Double]]` 型の値として与えられる．この行列の上三角化を行う函数 `triangular` を開発せよ．三角化後のリストには対角線の上および右上の値だけが入っているものとする．

例:

```
triangular( List( List(2,2,3,10),
                  List(2,5,12,31),
                  List(4,1,-2,1)) ) = List( List(2,2,3,10),
                                            List(3,9,21),
                                            List(1,2))
```

$$\begin{pmatrix} 2 & 2 & 3 & 10 \\ 2 & 5 & 12 & 31 \\ 4 & 1 & -2 & 1 \end{pmatrix} \triangleq$$

```
List( List(2,2,3,10),
      List(2,5,12,31),
      List(4,1,-2,1))
```

課題 3: solve.scala

連立方程式が `List[List[Double]]` の値として与えられた時，この解を計算する函数 `solve` を開発せよ．課題 2 で開発した `triangular` を使って良い．解は `List[Double]` の値とし，先頭から順に `x,y,z,···` の値として解釈する．また与えられる方程式は唯一の解を持つとしてよい．

例

$$\texttt{solve}\left(\begin{array}{l}\texttt{List(}\ \texttt{List(2,2,3,10),}\\ \qquad\ \texttt{List(2,5,12,31),}\\ \qquad\ \texttt{List(4,1,-2,1))}\end{array}\right) = \texttt{List(1,1,2)}$$

$$\texttt{List(}\ \texttt{List(2,2,3,10),}\\ \qquad\ \texttt{List(2,5,12,31),}\\ \qquad\ \texttt{List(4,1,-2,1))} \triangleq \left\{\begin{array}{rcrcrcr} 2\cdot x & + & 2\cdot y & + & 3\cdot z & = & 10\\ 2\cdot x & + & 5\cdot y & + & 12\cdot z & = & 31\\ 5\cdot x & + & 1\cdot y & - & 2\cdot z & = & 1\end{array}\right.$$

ガウスの消去法

1 次 n元連立方程式の解法として，ガウスの消去法がある．

1. 係数と定数から行列を作る
2. 作った行列を三角行列に変換する (三角化)
3. それぞれの変数が取るべき値を行列を下から辿ることで計算する

ガウスの消去法

1 次 n元連立方程式の解法として，ガウスの消去法がある．

1. 係数と定数から行列を作る
2. 作った行列を三角行列に変換する (三角化)
3. それぞれの変数が取るべき値を行列を下から辿ることで計算する

$$
\begin{aligned}
2 \cdot x + 2 \cdot y + 3 \cdot z &= 10 \\
2 \cdot x + 5 \cdot y + 12 \cdot z &= 31 \\
4 \cdot x + 1 \cdot y - 2 \cdot z &= 1
\end{aligned}
$$

を例として，ガウスの消去法を説明する．

係数と定数から行列を作る

$$\begin{aligned} 2\cdot x+2\cdot y+3\cdot z &= 10\\ 2\cdot x+5\cdot y+12\cdot z &= 31\\ 4\cdot x+1\cdot y-2\cdot z &= 1 \end{aligned}$$

は

$$\begin{pmatrix} 2 & 2 & 3\\ 2 & 5 & 12\\ 4 & 1 & -2 \end{pmatrix}\begin{pmatrix} x\\ y\\ z \end{pmatrix}=\begin{pmatrix} 10\\ 31\\ 1 \end{pmatrix}$$

と表せる．左辺の係数行列と右辺の定数行列を繋ぎあわせると以下を得る:

$$\left(\begin{array}{ccc|c} 2 & 2 & 3 & 10\\ 2 & 5 & 12 & 31\\ 4 & 1 & -2 & 1 \end{array}\right)$$

上三角化

行列を Mとするとき，M_iは i行目を，M_{ij}は i行 j列目を表すとする．$M_{(i+k)} = M_{(i+k)} - c_k M_i$を計算して $i+1$行目以降の全ての行で i列目の値を 0 にする．c_kは $M_{(i+k)i} - c_k M_{ii} = 0$を満たす実数である．

上三角化

行列を Mとするとき，M_iは i行目を，M_{ij}は i行 j列目を表すとする．$M_{(i+k)} = M_{(i+k)} - c_k M_i$を計算して $i+1$行目以降の全ての行で i列目の値を 0 にする．c_kは $M_{(i+k)i} - c_k M_{ii} = 0$を満たす実数である．

$$\begin{pmatrix} 2 & 2 & 3 & 10 \\ 2 & 5 & 12 & 31 \\ 4 & 1 & -2 & 1 \end{pmatrix} \xrightarrow[M_3 = M_3 - 2 \cdot M_1]{M_2 = M_2 - M_1} \begin{pmatrix} 2 & 2 & 3 & 10 \\ 0 & 3 & 9 & 21 \\ 0 & -3 & -8 & -19 \end{pmatrix}$$

上三角化

行列を Mとするとき，M_iは i行目を，M_{ij}は i行 j列目を表すとする．$M_{(i+k)} = M_{(i+k)} - c_k M_i$を計算して $i+1$行目以降の全ての行で i列目の値を 0 にする．c_kは $M_{(i+k)i} - c_k M_{ii} = 0$を満たす実数である．

$$\begin{pmatrix} 2 & 2 & 3 & 10 \\ 2 & 5 & 12 & 31 \\ 4 & 1 & -2 & 1 \end{pmatrix} \xrightarrow[M_3 = M_3 - 2 \cdot M_1]{M_2 = M_2 - M_1} \begin{pmatrix} 2 & 2 & 3 & 10 \\ 0 & 3 & 9 & 21 \\ 0 & -3 & -8 & -19 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 2 & 3 & 10 \\ 0 & 3 & 9 & 21 \\ 0 & -3 & -8 & -19 \end{pmatrix} \xrightarrow{M_3 = M_3 - (-1) \cdot M_2} \begin{pmatrix} 2 & 2 & 3 & 10 \\ 0 & 3 & 9 & 21 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

上三角化

行列を Mとするとき，M_iは i行目を，M_{ij}は i行 j列目を表すとする．$M_{(i+k)} = M_{(i+k)} - c_k M_i$を計算して $i+1$行目以降の全ての行で i列目の値を 0 にする．c_kは $M_{(i+k)i} - c_k M_{ii} = 0$を満たす実数である．

$$\begin{pmatrix} 2 & 2 & 3 & 10 \\ 2 & 5 & 12 & 31 \\ 4 & 1 & -2 & 1 \end{pmatrix} \xrightarrow[M_3 = M_3 - 2 \cdot M_1]{M_2 = M_2 - M_1} \begin{pmatrix} 2 & 2 & 3 & 10 \\ 0 & 3 & 9 & 21 \\ 0 & -3 & -8 & -19 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 2 & 3 & 10 \\ 0 & 3 & 9 & 21 \\ 0 & -3 & -8 & -19 \end{pmatrix} \xrightarrow{M_3 = M_3 - (-1) \cdot M_2} \begin{pmatrix} 2 & 2 & 3 & 10 \\ 0 & 3 & 9 & 21 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

変数の値を順に計算する

$$\begin{pmatrix} 2 & 2 & 3 & 10 \\ 0 & 3 & 9 & 21 \\ 0 & 0 & 1 & 2 \end{pmatrix} \triangleq \begin{cases} 2\cdot x + 2\cdot y + 3\cdot z = 10 \\ 0\cdot x + 3\cdot y + 9\cdot z = 21 \\ 0\cdot x + 0\cdot y + 1\cdot z = 2 \end{cases}$$

より，行列を下から順に計算してゆくことで解を得る:

- $1\cdot z = 2 \Rightarrow z = 2$
- $3\cdot y + 9\cdot z = 21 \equiv 3\cdot y + 9\cdot 2 = 21 \Rightarrow y = 1$
- $2\cdot x + 2\cdot y + 3\cdot z = 10 \equiv 2\cdot x + 2\cdot 1 + 3\cdot 2 = 10 \Rightarrow x = 1$

課題 2: triangular.scala

連立方程式の係数と定数から作った行列が
`List[List[Double]]` 型の値として与えられる．この行列の上三角化を行う函数 `triangular` を開発せよ．三角化後のリストには対角線の上および右上の値だけが入っているものとする．

例:

```
triangular( List(  List(2,2,3,10),
                   List(2,5,12,31),
                   List(4,1,-2,1)) ) =  List(  List(2,2,3,10),
                                               List(3,9,21),
                                               List(1,2))
```

$$\begin{pmatrix} 2 & 2 & 3 & 10 \\ 2 & 5 & 12 & 31 \\ 4 & 1 & -2 & 1 \end{pmatrix} \triangleq$$

```
List(  List(2,2,3,10),
       List(2,5,12,31),
       List(4,1,-2,1))
```

課題 3: solve.scala

連立方程式が `List[List[Double]]` の値として与えられた時，この解を計算する函数 `solve` を開発せよ．課題 2 で開発した `triangular` を使って良い．解は `List[Double]` の値とし，先頭から順に `x,y,z,···` の値として解釈する．また与えられる方程式は唯一の解を持つとしてよい．

例

$$\texttt{solve}\left(\begin{array}{l}\texttt{List(}\ \texttt{List(2,2,3,10),}\\ \qquad\ \texttt{List(2,5,12,31),}\\ \qquad\ \texttt{List(4,1,-2,1))}\end{array}\right) = \texttt{List(1,1,2)}$$

$$\texttt{List(}\ \texttt{List(2,2,3,10),}\\ \qquad\ \texttt{List(2,5,12,31),}\\ \qquad\ \texttt{List(4,1,-2,1))} \triangleq \left\{\begin{array}{rcrcrcr} 2\cdot x & + & 2\cdot y & + & 3\cdot z & = & 10\\ 2\cdot x & + & 5\cdot y & + & 12\cdot z & = & 31\\ 5\cdot x & + & 1\cdot y & - & 2\cdot z & = & 1\end{array}\right.$$