H22冬学期情報科学シケプリ ver 1.1

理科一類22組寺尾

このシケプリは講義で扱われた項目のうち、特に重要そうな概念の説明を目的としています。
講義内容すべてをまとめたわけではないので注意してください。
試験情報

1. 試験範囲：教科書第一章から第七章
2. 試験時間：90分
3. 持ち込み不可。

目次

1.Rubyの文法まとめ

1.算術演算子

+	加算
-	減算
*	乗算
/	除算
%	剰余
**	累乗
=	代入

注意
被演算子の片方が実数ならば演算結果も実数となる。

```
3 / 2 => 1
3.0 / 2 => 1.5
```

end

2.比較演算子

==	等しい
!=	等しくない
>	大なり
<	小なり
<=	以上
>=	以下

end

3.論理演算子

&& AND
|| OR
! NOT

end

4.条件分岐

if(条件)
条件が真のときの処理
else
条件が偽のときの処理
end

end

5.反復構造

1.for文

for i in 0..n
処理
end
注意
0..nの場合、最後のiの値がnとなる。
（ここを0...nと書くと最後のiの値は(n-1)となる。）

2.while文

while(条件)
処理
end

end

6.関数

def 関数名(引数１, 引数２,…)
処理
return 戻り値
end
注意
return文に到達するとプログラムはその時点で計算を終了し、
returnの後に書かれた式の値を返す。
returnが書かれていない場合は最後の行が戻り値となる。

end

7.nil

「何もない」というオブジェクト

end

8.配列

ルビーでの配列はすべて可変長。
取得方法
1.a = Array.new(3) => [nil, nil, nil]
2.a = [0,0,0] => [0,0,0]

配列に関する関数

length　　配列の長さを返す

ほかにもいろいろあるが説明がしにくいので下の例で理解してほしい。

```
a = [0,1,2]
a.length()   => 3
a[0]         => 0
a[1]         => 1
a[3]         => nil
a[4] = 5
a            => [0, 1, 2, nil, 5]
a[0] = [2.4, 3, "this"]
a            => [[2.4, 3, "this"], 1, 2, nil, 5]
a[0..2]      => [[2.4, 3, "this"], 1, 2]              #a自体は変わらない。
a << 3.14    => [[2.4, 3, "this"], 1, 2, nil, 5, 3.14] #a自体が変わる。
a = [0,1,2]
b = [3,4,5]
c = a+b      => [0,1,2,3,4,5]
```

end

9.文字列

文字の配列だと考えればよい。使える関数も同じ。

end

end

2.再帰

1.定義

関数 f が再帰的とは f の定義の中に f 自身が使われることである。

end

2.再帰的関数のオーソドックスな書式

```
def f( n , a1, a2, ...)
    if 境界条件（たとえばn == 0）
        処理
        戻り値
    else
        処理
        f( n - 1, a1', a2' ,... )
        処理
        戻り値
    end
end
```

3.注意事項

・当たり前だが、再帰的に呼び出されるものはいつか境界条件に至らなければならない。

・また理想的には再帰の深さはいくらでも深くなれるが、
現実には深すぎる再帰はエラーとなる。（Rubyでは3千（笑）程度。Cは30万程度。）

・さらに、再帰はメモリリソースを食うので、繰り返しで楽に書けるようなものを

再帰で書くことはあまりよくない。
・関数の中で2回以上再帰呼び出しをする関数は後述する動的計画法を用いることが多い。
end

4.再帰を書く上でのコツ
深いことは考えず、 f が何を表すのかを考え、f (n)とf (n - 1)の関係を考える。
f(0), f(1),f(2)など値の小さい方から考えるとわかりやすい。
end
end

３.（時間）計算量
1.定義
アルゴリズムから見積もられる、入力の大きさと計算時間増大の大まかな関係のこと。
o()で表す。
1. 定数を引数とする処理、および標準関数などの計算量はo(1)。
2. 計算量o(k)の計算をn回繰り返す処理の計算量はo(nk)
3. o(f(n)) = o(k f(n)) (kは定数)
4. o(f(n)) の処理のあとにo (g(n)) の処理を行う場合の計算量はo(max(f, g))
ここでのmaxはnを十分に大きくした場合での大小関係である。
end

2.計算量の大小
o(logn) < o(√n) < o(n) < o(nlogn) < o(n^2) <<（越えられない壁）<< o(2^n)
左の方が「よい」アルゴリズム。
end

3.計算量の定め方
入力するデータの性質によって計算時間が大きく異なることがある。
このような場合、計算量をどのように定義するかにはいくつか種類がある。
代表的なものは以下の二つである。
1. 同じ長さの入力の計算量の平均とする。
2. もっとも時間がかかる入力の計算量とする。
今回の講義では後者をアルゴリズムの計算量と定義しているようである。
end

4.注意
計算量が小さいアルゴリズムが必ずしも最適なプログラムとは限らない。
たとえば、扱うデータが小さい場合には定数時間の処理が無視できない。
また、時間計算量が小さいかわりに使用するメモリが多いアルゴリズムもある。
使用するアルゴリズムは入力の大きさや種類に応じて総合的に判断されるべきである。
end
end

４.誤差

1.コンピュータ上での実数データの表現

コンピュータ上では実数は主に浮動小数点表現によって表現される。
詳細は教科書120ページ付近を読んでほしい。（前期の情報の講義でもやったはず）
重要なのはあくまで実数の近似値をとっているに過ぎないということ。

end

2.丸め誤差

有限の桁数で近似しているために起こる誤差。
電卓で1.0 / 3.0 * 3が0.99...9になるようなもの。

end

3.桁落ち誤差

非常に近い2つの数の差を計算したときに有効数字が減ることによって起きる誤差。
微分計算などで起きる。

end

4.情報落ち誤差

大きな数と小さな数を足し算したときに、小さな数が大きな数の有効桁数より
小さい場合、足し算しても値が変化しないことによって生じる誤差。
積分計算などで起きる。

end

5.打ち切り誤差

無限級数で表される数を有限の足し算で打ち切るために生じる、真の値との誤差。
これは実数の表現法とは関係なく発生する。

end

end

５.動的計画法

1.動的計画法とは

再帰的な関数で、定義中で2回以上再帰呼び出しを行うものは、
呼び出し回数が指数的に増加し、計算時間が急激に増大してしまう。
このような関数に対して、一回計算した値を配列に記憶し
それ以降の計算はその配列を参照することで無駄な計算を防ぐことが動的計画法である。
この講義においては先に配列を用意し、それを小さい方から埋めていくという方式が
とられている。

end

2.書き方

フィボナッチ数列を求める関数を例に説明する。
まず再帰的なプログラムを考える。

```
def fib( n )
        if ( n == 0 || n == 1)
                1
        else
                fib( n - 1) + fib( n - 2)
        end
```

```
        end
        つぎにこれを動的計画法を用いたプログラムに書き換える。
        まず、必要な大きさの配列fibを用意する。
        この配列の添え字nの要素にfib(n)を小さい順に入れていく。

        fib[0] とfib[1]は上のプログラムにおける境界条件なので1を代入する。
        次に反復構造を用いてのこりを埋めていく。
        具体的には上のプログラムにおけるfib( )をfib[ ]に書き換え、
        戻り値を配列の要素に代入すればよい。
        具体的には以下のようになる。

        def fib_dp( n )
            fib = Array.new( n )
            fib[0] = fib[1] = 1
            for i in 2..n
                fib[i] = fib[i -1] + fib[i-2]
            end
            fib[n]
        end
    end
end
```

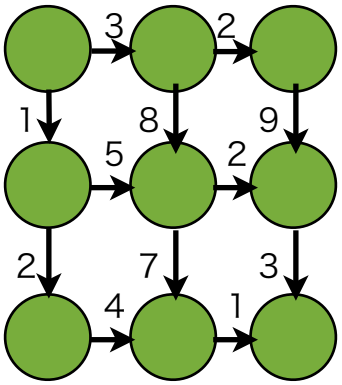
６.練習問題

1.実数a,bと自然数dと関数f(x)が与えられたときに区間[a,b]に存在する
方程式f(x) = 0の解（ただしただ一つの解を持つことは前提とされているものとする）を
2^(-d)の精度で求めるアルゴリズムの1つに次のようなものがある。

a1 = a, b1 = bとする。
m = (a1 + b1)/2とし、f(m) を計算する。
f(a1)*f(m) <= 0ならばa2 = a1, b2 = mとする。
f(b1)*f(m) <= 0ならばa2 = m , b2 = b1とする。
その後同じようにしてa3,a4,...,b3,b4,...を計算する。
bn - an < 2^(-d+1)となったら計算を終了し、(an+bn)/2を答えとして返す。

(1)このアルゴリズムを実装したプログラムを反復構造を用いたものと、再帰を用いたもの
の両方で書け。ただし、関数fの具体的な中身は定義しなくてもよい。
(2)このアルゴリズムの計算量を評価せよ。ただしf(x)の計算量はo(1)とする。
(3)実数表現として倍精度浮動小数点表現を用いた場合、 d をある数以上にすると
計算が終わらなくなる。a=0.5, b=1.0のときその数をもとめ、
なぜ計算が終わらなくなるのかを説明せよ。

2.M行N列の格子状の有向ネットワークを考える。M*N*2の大きさの3次元配列mapが与えられて、
map[i][j][0]には(i , j)から(i +1, j)への距離(>0)がはいっていて、
map[i][j][1]には(i , j)から(i , j+1)への距離(>0)が入っているものとする。
ただし、(i, j)からは(i, j-1), (i - 1, j) には進めないものとする。
このとき(0, 0)から(p, q)の最短距離distance(p, q, map)は
次のように再帰的に定義される。




```
def distance( p, q, map )
    if ( p == 0 && q == 0 )
        return 0
    end
    if ( p == 0 )
        return distance( p, q - 1, map ) + map[p][q-1][1]
    end
    if ( q == 0 )
        return distance( p - 1, q, map ) + map[p-1][q][0]
    end
    d1 = distance( p - 1, q, map ) + map[p-1][q][0]
    d2 = distance( p, q - 1, map ) + map[p][q-1][1]
    return min( d1, d2 ) #d1とd2の小さい方を返す。
end
```

このプログラムがなぜ最短到達時間を計算できるのか説明し、
動的計画法を用いたプログラムに書き換えよ。

end

7.解答

1

(1)反復構造を用いたもの

```
def solve( a, b, d )
    m = ( a + b ) / 2.0
    while( b - a < 2**(-d+1) )
        m = ( a + b ) / 2.0
        if ( f( a )* f ( m ) <= 0 )
            b = m
        else
            a = m
        end
    end
    m
end
```

再帰を用いたもの

```
def solve( a, b, d )
    m = ( a + b ) / 2.0
    if ( b - a < 2**(-d+1) )
        return m
    end
    if ( f( a ) * f( m ) <= 0 )
        solve( a, m, d )
    else
        solve( m, b, d )
    end
end
```

(2)計算量はo(d*log(b-a))

(3)d = 54。倍精度浮動小数点表現では仮数部に52ビットを使う。
a,bが等しくなることはなく、ともに1.d1d2...d52*2^(-1)と表されるため、

b-aの最小値は2^(-53)である。したがってdが54以上の時、ループあるいは再帰を
抜けるための条件を満たすことはなく、プログラムが停止しない。

2

(1)(0,0)から(p,q)までの最短到達経路は(p-1,q)か(p,q-1)のどちらかを必ず通る。
したがって、(p, q) までの最短距離は
((p-1, q)までの最短距離 + (p-1, q)から(p, q)までの距離)と
((p, q-1)までの最短距離 + (p, q-1)から(p, q)までの距離)
の小さい方である。ただ、p=0（q=0)の時は(p-1, q) ((p, q-1))が存在しないため、
例外的に処理している。

動的計画法を用いたプログラムの例は以下の通り

```
def distance_dp( p, q, map )
        distance = make2d( p+1, q+1 ) #2次元配列を取得
        distance[0][0] = 0
        for i in 1..p
                distance[i][0] = distance[i-1][0] + map[i-1][0]
        end
        for i in 1..q
                distance[0][i] = distance[0][i-1] + map[0][i-1]
        end
        for i in 1..p
                for j in 1..q
                        d1 = distance[i-1][j] + map[i-1][j][0]
                        d2 = distance[i][j-1] + map[i][j-1][1]
                        distance[i][j] = min( d1, d2 )
                end
        end
        distance[p][q]
    end
  end
end
```