電車の中で読もうと思えば読めないことはない情報科学シケプリ

情報科学、通称 JK のシケプリです。振動波動論、通称しんぱっぱと日程がかぶってるので JK にあまり時間を割けないと思います。ので、テストに即した軽く読めるものに仕上げてみました。各章のエッセンス、リファレンス、テクニックなどが盛り込まれています。

第 1 章 数の計算と関数

この章は、コンピュータとの対話、数式の計算、変数、関数の定義、ファイルに保存した関数定義から構成されています。過去の期末の問題をみてもこの章の内容に関する直接の出題はありません。分かってて当然、といったスタンスでしょう。

割り算などの演算をした後の**型**に注意しましょう。**整数/整数=整数**となります。
数学関数を呼び出すには include(Math) をファイルの最初に宣言しておかなければなりません。
変数は、必ず小文字から始めます。
関数定義の雛型は、

```
include(Math)
def func1(hoge1,hoge2)#func の中のものを仮引数といいます。func(仮引数 1,仮引数 2)ですね。
        hoge1+hoge2 #オブジェクトが単体でおかれると、これは返り値とみなされます。
                    #関数の呼び出しもとへ渡す値ですね。
end

def func2() #仮引数がなくてもカッコは書きましょう。
        func1(10,22)**2
end
```

です。関数の中で定義された変数は、特別の修飾子が付いていない限りは、局所変数（ローカル変数）といって、その関数の中からしか呼び出すことができません。以下、非常に大切になってくるので、局所変数のスコープ（有効範囲）には細心の注意を払うようにしましょう。基本的にスコープ内では、変数は保持し続けますが、一旦スコープを抜けると変数は消滅します。
hoge.rb ファイルの中で、piyo.rb ファイルに定義した関数を呼び出したい場合は、`load("./piyo.rb")` といった風にファイルの先頭に宣言します。

第 2 章 配列による画像の表示

この章は、画像の表現、画像の操作、カラー画像の表現から構成されています。この章の内容はみなさん慣れたものでしょう。配列に関する事項だけ確認しておきましょう。

ruby には配列を実装する手段として Array クラスのインスタンスを作るという方法が提供されています。ただし、２次元以上の配列は標準では用意されていないので、実装するには新たに作る必要があります。

```
def make1d(n)
  a = Array.new(n)
  for i in 0..(n-1)   #配列のインデックスは０オリジンです。   これ大事。
    a[i] = 0
  end
  a
end

def make2d(height,width)
  a = Array.new(height)
  for i in 0..(height-1)
    a[i] = make1d(width) #いちいち別の make1d 配列を用意していることに注意。
  end
  a
end
```

２次元以上の配列の呼び出し方は少しややこしいですね。要は配列の各要素に配列をぶちこんだ、ってことなんですから、例えば a[4][10] という風に呼び出すと、a という配列の５番目の要素は配列だけれども、その配列の 11 番目の要素！っていう感じに解釈できます。

第 3 章 条件分岐と繰り返し

この章は、条件分岐、真偽値を与える論理計算、文字列、繰り返しによる画像の作成から構成されています。条件分岐に関しては簡単なので試験には出ないとおもいます。繰り返しに関しては「何回この行が実行されるか？」といった類の問題がみられますので、注意が必要です。

3.1 条件分岐

まずは雛型をば。

```
def max(a,b)
  if a>b  #評価が true の時実行される
    a
  elsif b>a
    b
  else
    a
```

```
  end
end
#あえて冗長な書き方をしていますが、elsif の使用例だと思ってください。

#授業では扱いませんでしたが、知っていると便利かもしれないので一応載せます。
#case−when 構文 case の後に評価したい物を置きます。その値に応じて実行する処理が異なるのがこれです。
def days_of_month(year,month)
  case month
  when 1 then return 31
  when 2 then days_of_february(year)
  when 3 then return 31
  when 4 then return 30
  when 5 then return 31
  when 6 then return 30
  when 7 then return 31
  when 8 then return 31
  when 9 then return 30
  when 10 then return 31
  when 11 then return 30
  when 12 then return 31
  end
end
```

条件分岐には評価する対象が必要です。
&&,||といった類の論理演算子は先学期の情報でやりましたね。

文字列は、ruby においては String クラスのオブジェクトとして管理されています。String クラスには、演算子オーバーロードと呼ばれる＋や－といった関数が用意されていますので、文字列の足し算とか引き算とかができるようになるわけです。

3.2 繰り返し

```
def func1(a) #配列を仮引数とします。
  for i in 0..a.length()−1 #a の配列の長さ−1としてるのは、配列が 0 オリジンだからですね。
    a[i] = a[i]∗2
  end
end

def func2()
  while true
    print "hoge"
  end
end
#実行することはお勧めしません。
#while 文の評価式の中身が true なら実行し続けるということです。
```

繰り返しは、入り子にもできます。２重ループとかがそれですね。
ここで、for ループの中のある行が何回実行されているのか、といったようなことが試験によく出ています。簡単のように思えますが、for ループの中に条件式とかが紛れ込んだりすると結構ミスします。

第 4 章 関数から計算へ

この章は、繰り返しによる反復計算の定義、再帰による反復計算の定義、配列・文字列と繰り返しで構成されています。繰り返しによる反復計算は、for 構文と変数のスコープがしっかり理解できていれば問題ないので、再帰を中心に説明します。

再帰とは、端的にいえば、ある関数の定義の中にその関数を用いるものです。

```
# sum of the numbers from 1 to n
def sum(n)
  if n >= 2
    sum(n -1) + n
  else
    1
  end
end

def fibr (k) #Fibonacci数を再帰的に求めるプログラム
  if k==0 || k==1
    1
  else
    fibr (k -1) + fibr (k -2)
   end
end
```

この例で分かるように、sum 関数の中に sum 関数を使用しています。漸化式をそのままプログラミングにできると思えば概ね差し支えありません。ですから、初期条件というものが必要になってきます。これを落とすとエラーが出ますのできちっと初期条件を書くようにしましょう。

第 5 章 アルゴリズムと計算量

この章は、Fibonacci 数を求めるアルゴリズム、計算時間の違いと計算量、行列を用いた Fibonacci 数のアルゴリズム、整列のアルゴリズムで構成されています。いままでの 4 章は文法を扱うといった面が強かったですが、この章からはアルゴリズムに焦点を当てています。特にこの章の計算量は、毎年のようにテストに出ていますので要注意です。

5.1 アルゴリズムから計算時間を見積もる

再帰的に Fibonacci 数を求めるというのは前章でもやりました。今度は for ループで作ってみましょう。

```
def fibl (k)
  f = 1
  p1 =1
  for i in 2..k   #ただ中身を繰り返すという意味
    p2 = p1   #fib(i-2)
```

```
    p1 = f   #fib(i−1)
    f = p1 + p2   #fib(i)
  end
  f #fib(k)
end
```

どういう仕組みなのかは、実際に変数の値の変化を追っていけば理解できるでしょう。さて、ここでこの計算にかかる時間を考えてみましょう。ＰＣの処理速度は一定として、計算にかかる時間と計算する量は比例します。なので一般的にアルゴリズムを評価する手段として、”計算量”を求めそこから計算時間を見積もる、ということをします。計算量というのは、実際に行う演算の回数をもっとも簡単に近似したものです。要は、演算回数に最も影響を与えている項のことですね。上にあげた Fibonacci 数を求めるプログラムで実際に計算量を求めてみましょう。

まず、再帰的なアルゴリズムの `fibr(k)` を計算量を考えてみましょう。`fibr(k)` を求めるためには再帰によって `fib(k-1)` と `fib(k-2)` を求める必要があります。`fib(k-1)` と `fib(k-2)` もやはり再帰によって求められるので、計算量も Fibonacci 数列になりますね。なので計算量は (黄金比)^k になっていることがわかります。

これに対して、for ループによる計算量を求めてみましょう。これは for ループを $k-1$ 回回すだけなので計算量は k となっています。（当然ですがかなり近似してます。）

以上の計算量を、O(びっぐおー) を用いて表現します。つまり、

$$\texttt{fibr(k)} \text{ は } O((\text{黄金比})^k), \quad \texttt{fibl(k)} \text{ は } O(k)$$

と書けます。

このような、Fibonacci 数を求めるアルゴリズムは自由な要素がない（Fibonacci 数を求めるためには k=1 まで実行しなければならない) ので厳密に計算量を計算することもできますが、ある配列から目的の値を持ったものを探す、といった演算する回数が定まっていない場合に対しても計算量を定義します。

この場合、最悪の場合に計算量を代表させるのが一般的です。（例えば、100 個のリストの中からＡ君を見つけようとしたときに、100 番目にみたリストにＡ君の名前がある場合などです。）実は、以上の 2 つのアルゴリズムよりも少ない計算量で求めることができます。行列を使う方法が教科書で紹介されています。とは言え、これだけだと一見行列を n 乗するので下手すると数え上げるよりも演算回数が多そうですね。そこで、行列の掛け算をさせるアルゴリズムを考えます。なんと、$O(\log_2 k)$ でできるんですね。これは k を 2 進数表示したときに出る桁数分で計算量が代表できるということです。これは、行列の 2 乗の 2 乗の計算が 2 回でできることに起因しています。具体的にいえば以下のアルゴリズムに従って行列の n 乗を求めています。

$$Q^n = \begin{cases} E & (n = 0) \\ (Q^{n/2})^2 & (n \text{ は 2 以上の偶数}) \\ Q \times Q^{n-1} & (n \text{ は奇数}) \end{cases}$$

例えば Q^{19} を例にとってみてみましょう。

$19_{(10)} = 10011_{(2)}$ なので

$$Q^{1\times 2^4} \times Q^{0\times 2^3} \times Q^{0\times 2^2} \times Q^{1\times 2^1} \times Q^{1\times 2^0}$$

です。計算量においてはこのうち $Q^{1\times 2^4}$ に代表されますから、このアルゴリズムの計算量は $O(\log_2 k)$ ということになります。

5.2 整列

ランダムな数列を降順なり昇順なりに並べ替えることを整列といいます。こういうのは５２枚ばらばらなトランプを買って来た状態に戻すのをコンピュータにやらせると思えばいいですね。教科書に載っているのは**単純整列法**と**併合整列法**のみなのでこれらをマスターすれば大丈夫かと思います。

5.2.1 単純整列法

単純整列法はいたって単純、まず配列をすべて見渡してそのうち一番小さい値を配列の一番前に持ってくるというものです。トランプの例でいうと、まず山の中からスペードの 1 を探し、そのあとにスペードの 2 を探し・・・という感じですね。このまんまプログラムにすると別の配列を用意する必要があるし、同じ評価を何回も行うことになりますので、少し工夫したものを考えます。すなわち、「元の配列の中で 1 番小さな数を見つけそれを配列の 0 番目と入れ替える。次に配列の 1 番目以降で 1 番小さな数 (つまり全体では 2 番目に小さな数) を見つけ、配列の 1 番目に場所に置く」という作業を繰り返します。これをプログラムにすると

```
def min_index(a,i) #a[i]以降で最小の値が現れる場所を探して、その場所を返します。
  min = a[i]   #仮に、一番最初を最小値としてます
  min_num = i #そのインデックスです
  for k in (i+1) .. (a.length-1)
    if a[k] <= min    #あとあと上で見た最小値より小さいのが見つかれば塗り替えるということですね
      min = a[k]
      min_num = k
    end
  end
  min_num
end

def simplesort (a)
  for i in 0..(a. length() -1)
    k = min_index(a,i)   #i 以降で一番小さい値のインデックスを k に格納します
    v = a[i]   #破壊的にならないようにインデックス i にあった値を保存しておきます。
    a[i] = a[k]   #インデックス i に k の値をいれます
    a[k] = v   #もともとあったインデックス i の値を代わりにインデックス k にいれます。
  end
  a
end
```

この方法の演算回数は $\frac{1}{2}n(n-1)$ なので、計算量は $O(n^2)$ となります。しかし、これはまーったく現実的な計算量ではありません。なので、そこそこましな併合整列法を考えます。

5.2.2 併合整列法

簡単に言うと、あらかじめ順序良く (well-ordered) 並んでいる二つの配列があるとそれらを併合 (merge) して順序良く並べることができる、という方法を応用したものが併合整列法です。トランプの例でいうと 1 列に並べられたばらばらなトランプを左から二つずつ併合し、できた 26 組の山を左から順にそれぞれ 2 組ずつ併合、できた 13 組を・・・といった感じです。

```
load("./is_even.rb")
#便利なんでポインタという表現を多用してますが、例の”ポインタ”とは何の関係もないので。
def merge(a,b)  #2つの配列を与えてその併合した配列を返す。
  c = Array.new(a.length()+b.length())  #まず、併合した後に値を格納する配列を作る。
  ia = 0  #カウンターをセット  イマココみてる！っていう矢印を表してると思えばよかろう
  ib = 0  #カウンターセット  これもおなじ
  ic = 0  #カウンターセット  次の値はここに格納するっていう矢印
  while ia < a.length() && ib < b.length() #まだポインタが配列の中をさしているか。
    if a[ia] < b[ib] #ポインタが指している値を不等評価
      c[ic] = a[ia]
      ia = ia + 1 #カウンターを一個次へ
      ic = ic + 1
    else
      c[ic] = b[ib]
      ib = ib + 1
      ic = ic + 1
    end
  end
  if a.length()−ia >= 1    #余ってる値をすべてそのまま c へ
    for k in (ia+ib) .. (c.length()−1)
      c[k] = a[k−ib]
    end
  else
    for k in (ia+ib) .. (c.length()−1)
      c[k] = b[k−ia]
    end
  end
  c
end

def mergesort(a)
  n = a.length()  #トランプの例でいうと山の数に対応している
  from = Array.new(n)
  for i in 0..(n−1)
    from[i] = [ (a[i]) ] #配列に配列を格納していることに注意。n 行 1 列の行列を作っていることになる。
  end
  while n > 1 #山が 2つ以上あるなら
    to = Array.new((n+1)/2)  #まず、併合後の配列をつくる。この変数のスコープに注意。
    for i in 0..(n/2−1) #実際にマージする回数
      to[i] = merge(from[i*2], from[i*2+1])  #マージ  インデックスが小さいほうに詰めていく
    end
    if !is_even(n) #もし奇数なら
      to[(n+1)/2−1] = from[n−1] #一つ余るので最後につけ足しておく。
    end
    from = to #今マージしたやつを改めてマージ元にする
    n = (n+1)/2 #山の数  偶数でも奇数でも山の数は同じ
  end
  from[0]  最後に一つにまとまったもののindex0 が求める配列
end
```

では、この計算量を考えてみましょう。簡単のため、トランプの枚数が $n = 2^e$ だとしましょう。i 回目の繰り返しの時、各山の枚数は 2^{i-1} 枚なので、マージの際に行う演算回数は 2^i です。これを山の数/2 回行うことになるので、この一回の操作における演算回数は $2^i \times 2^{e-i} = 2^e = n$ 回です。この操作を行う回数は先ほどの行列と同じように $\log_2 n$ なので計算量は $O(n \log n)$ です。

第 6 章 数値計算

この章は、数値積分、乱数と Monte Carlo 法、実数データ表現と誤差、連立 1 次方程式、CT スキャンで構成されています。・・・正直あまり時間がないので過去問に出た Monte Carlo 法、Gauss-Jordan、応用問題となっている CT スキャンはここでは省略します。過去問を見てください。このシケプリでは、実数データ表現と誤差を中心にします。

6.1 実数データ表現

我々の用いている 10 進数はコンピュータにとってやさしくありません。なのでコンピュータにやさしい 2 進数で数を表記する手段として、IEEE754(あいとりぷりー）という規格が用いられています。この規格では、少ないメモリで非常に大きな数と非常に小さな数を浮動小数点表示という方法で表現でき、また、ビット数的にも OS との相性が良かったりします。まあ、御託はさておき、出そうな情報だけ要約して。この規格では、単精度表現と倍精度表現とがあります。単精度表現とは、32bit を符号部 (s)、指数部 (c)、仮数部 (m) の 3 つに分けて数を表現する方法で、それぞれのパートには 1bit, 8bit, 23bit が割り当てられています。倍精度ではこれがそれぞれ 1bit, 11bit, 52bit が割り当てられるだけで表現法自体は単精度と同じです。32bit 列を前から順に符号、指数、仮数に分け以下のように数を表現します。

$$(-1)^{s_{(2)}} a.d_1 d_2 \dots d_{m(2)} \times 2^{d'_1 d'_2 \dots d'_{c(2)} - b}$$

ただし、この式の中の b はバイアス値と呼ばれ、単精度では 127、倍精度では 1023 です。この値は指数部に依存し、$b = 2^c/2 - 1$ です。

符号は $s = 0$ なら正、$s = 1$ なら負、
指数は 1 から 8bit 目を正の整数として、そこからバイアス値を引いた値、
仮数は 9bit 目から 31bit 目までを 2^{-1} から 2^{-23} に対応させ、さらに 1 を足した値となります。
具体的に計算するのがいいと思うので練習問題 6.5 をみてみましょう。

練習 6.5 (浮動小数点数の表現)

a) 次のビット列は IEEE 754 規格の単精度浮動小数点数である。どのような値を表わしているかを求めて 10 進数で書け。$\overset{s}{1}\overbrace{11111110}^{d'_1,\dots d'_8}\overbrace{11000000000000000000000}^{d_1 \dots d_{23}}$

b) 次の数を IEEE 754 規格の単精度浮動小数点数で表わすときの $s, d'_1, \dots d'_s, d_1 \dots d_{23}$ をビット列で書け。
   (a) 1.0
   (b) $3145728_{(10)}$
   (c) $5/65536_{(10)} (\simeq 7.62939453125 \times 10^{-5})$

解答例 6.5

a) 符号、指数、仮数に注意して書き下すと

$$(-1)^1 \times 1.11000000000000000000000_{(2)} \times 2_{(10)}^{11111110_{(2)}-127}$$

となる。すなわち、$1.75 \times 2^{127} = 2.894802 \times 10^{38}$

b) (a) $1.0 = +1.00000000000000000000000 \times 2^{01111111-127}$より 00111111100000000000000000000000

(b) $3145728 = +1.10000000000000000000000 \times 2^{10010100-127}$より 01001010010000000000000000000000

(c) $5/65536 = 5 * 2^{-16} = +1.01000000000000000000000 \times 2^{01110001-127}$より 00111000101000000000000000000000

手計算なので間違ってるかもです。

説明 6.5

a) はまあいいでしょう。問題となるのは b です。10 進数を 2 進数に直す方法として数をひたすら 2 で割っていき、余りを隣に書いていくというものがあります。

$$\begin{array}{ll} 49 & \cdots 1 \\ 24 & \cdots 0 \\ 12 & \cdots 0 \\ 6 & \cdots 0 \\ 3 & \cdots 1 \\ 1 & \end{array}$$

これを下から反時計回りにみていくと 2 進数表示になります。

6.2 誤差

この教科書では 4 つの誤差について触れています。

- 丸め誤差
  数値を、ある桁で端数処理することによって現れる誤差である。例としては、ある数（10 進数表記では正確）を IEEE7754 規格の単精度浮動小数点表示した場合に仮数部の有効数字 24 桁で近似する際に生じる。
- 情報落ち誤差
  演算する 2 数の有効桁の相違によって生じる誤差である。例としては、大きな値に小さな値を足す場合、小さな値が有効桁の範囲外になってしまうときに生じる。
- 打切り誤差
  無限和を、不十分な有限和（さらに有限和をとり続けると近似値が変化する）で近似しようとする際に現れる誤差である。例としては、テイラー展開などを求める際に生じる。
- 桁落ち誤差
  有効桁数が減ることによっておこる誤差である。例としては、非常に近い 2 数の引き算を行う際に有効桁数が減少してしまうことによって生じる。

第 7 章 パターン認識

この章は、パターン認識の特徴、アラインメント、再帰によるアラインメント、動的計画法、動的計画法によるアラインメントで構成されています。アラインメントは 07 年に出題されているので、出ない気がします・・・。

第 8 章 レコードとオブジェクト

この章は、複数の値をまとめるレコード、値と操作をまとめるオブジェクト指向で構成されています。クラス、インスタンス、メンバなどの OOP(Objective Oriented Programming：オブジェクト指向プログラミング) について触れることにします。

まずは、この章で新出の文法をまとめたコードを

```
class Line #クラスの宣言
  attr_accessor ("p0", "p1")     #インスタンス変数（メンバ変数）インスタンスにリンクされている

  def initialize(q,r)         #初期化メソッドインスタンスが生成されたときに自動的に呼び出される。
    self.p0 = q   #self.メンバ変数でそのインスタンス自身のメンバ変数を指す
    self.p1 = r
  end

  def draw(a)         #インスタンスメソッド　インスタンスにリンクされている
    n = max(abs(self.p1.x − self.p0.x),
    abs(self.p1.y − self.p0.y))
    for i in 0..n
      p = self.p0.interpolate(self.p1, i∗1.0/ n)   #このように.を重ねて使うことができる
      p.draw(a) #インスタンスメソッドを呼び出している
    end
  end

  def turn(theta)
    self.p0 = self.p0.rotate(theta)
    self.p1 = self.p1.rotate(theta)
  end
end

class Bezier < Line       #Line クラスを継承している
  attr_accessor ("p0", "c", "p1") #インスタンス変数

  def initialize(q,r,s)
    super(q,s)         #継承したクラスの同じ名前の関数を呼び出している。
    self.c = r
  end

  def draw(a)
    n = 10
    prev = self.p0
    for i in 1..n
```

```
      t = i*1.0/ n
      q0 = self.p0.interpolate(c,t)
      q1 = self.c.interpolate(p1,t)
      r = q0.interpolate (q1,t)
      Line.new(prev,r).draw(a) #このようにその場で作成したインスタンスのメソッドを呼び出すこともできる
      prev = r
    end
  end

  def turn(theta)
    super(theta)
    self.c=self.c.rotate(theta)
  end
end
```

上でテストに出うる文法は網羅しているはずです。ここからはそれぞれの単語について少し補足を加えることにします。

まず、クラスですが、理解をやさしくするために Triangle クラスを考えてみましょう。このクラスが用意するべきものはなんでしょうか。まず、三角形は頂点のデータが必要ですので、インスタンス変数として p0, p1, p2 を用意します。色なんかもインスタンス変数として用意しておけばよさそうですね。あとはこの三角形の面積とかを返すメソッドとかがあれば便利そうですね。他にもこの三角形の法線ベクトルを返すメソッドとかもあれば何かと便利そうです。このように、”三角形”というものをインスタンス変数とインスタンスメソッドによって特徴づけることこそがクラスの定義を書くということです。ここで話に出てきた、インスタンスとは何なのでしょうか。答えは簡単です。この概念としてある三角形を実体（インスタンス）化したもののことです。我々の頭の中には、三角形というクラスが用意されていて、そのインスタンスを作成するという行為が紙に鉛筆で三角形を書くという行為に相当するのかもしれませんね。こう考えると、インスタンス変数である p0,p1 などの頂点は三角形を描いて初めて”見る”ことができますので、これらはインスタンスに結びつけられているものと捉えることが大事です。この授業では使いませんでしたが、クラス変数、クラスメソッドという概念もありこちらは、三角形という我々の頭の中にある概念そのものに結び付けられたものと考えることができます。例えばクラス変数としては、三角形の頂点の個数などがあげられるでしょう。これは別に実体化しなくても 3 個というのはわかりますからね。インスタンスメソッドも同様にして、インスタンスに結び付けられています。なので、クラスから直接インスタンスを呼び出そうとするのは、クラスメソッドでない限りエラーが出ます。あくまでインスタンスメソッドはインスタンスに結び付けられたものなので、一度 new などでインスタンス化してから呼び出さなければなりません。

初期化メソッドに関しては、クラスからインスタンスが作成されたときに最初に呼び出されるメソッドのことです。先ほどの例に当てはめると、例えば私たちが数学の問題を解こうとしているときに三角形の 3 点の情報が与えられたとします。我々はこの 3 点の情報をもとに三角形を描きます。この三角形を作成する際に、最初にインスタンスに渡される情報こそがインスタンスメソッドの仮引数となっています。

継承は、OOP の基本概念のひとつで、非常に重要なものです。上の例でも出てきましたが、これがあるから同じようなメソッドを繰り返し描く必要がなくなるというわけですね。ちなみにこの継承元のクラスを親クラス、継承したクラスを子クラスといいます。そして、子クラスは直属の親クラスだけではなくすべての親クラスの変数とメソッドが継承されています。ここで、子クラスで親クラスと同じメソッド名を持つ場合があるでしょう。この時は、子クラスで定義したメソッドが優先されます。つまり親クラスのメソッドが隠されるわけです。では、子クラスで一旦定義してしまうと親クラスで定義できないのでしょうか。そんなことはありま

せん。そのために、super キーワードが用意されているわけです。

とまあ上で用いられている概念、語句は上のような感じです。これでクラスに関しての説明は一通りできるはずです。

さて、OOP の概念には 3 本柱があります。継承はそのうちの一つです。他にも多態性（ポリモーフィズム)、カプセル化があります。子のごく説明は過去に出題があるので、出ることはないと思いますが、一応言葉だけは確認しておいた方がいいでしょう。07 年の問題と解答例を下に示します。

2007 年共通問題 4
(a) カプセル化 (情報隠蔽でもよい)、(b) 継承 (インヘリタンス)、(c) 多態性 (ポリモルフィズム) のそれぞれについて、
(ア) どのような概念であるか、
(イ) どのような問題を解決するために導入されたか、
を、それぞれ 3 行以内で簡潔に説明せよ。

解答例

1. カプセル化
   ア）データとそれを操作する手続きを一体化して「オブジェクト」として定義し、オブジェクト内の細かい仕様や構造を他のソフトウェアなどからから隠蔽すること。
   イ）カプセル化を用いずにオブジェクト内の機密情報へのアクセスを制御することを容易にし、情報を保存するために使用するデータの形式を変更したくなった時の手間を省くため。
2. 継承
   ア）一つのクラスによって既に定義されているメソッドや変数をもとに別のクラスを定義することを可能にすることにすること。
   イ）似たような処理を行うクラスを改めて 1 から定義しなおす必要がなく、またその意味において、ソフトウェアの再利用が促進される。
3. 多態性
   ア)「1 つのインターフェイス、複数の実装」と表現されるメカニズムであり、メソッドや定数などについてそれらが複数の型に属することを許すという性質を指す。
   イ）非オブジェクト指向プログラミング言語においては、異なった動作を実現するためには異なった名前を用いる必要があったが、これを同じ名前で実装できるようになった。

第 9 章 再帰データ構造

この章は、リスト構造、木構造と再帰で構成されています。07 年に出ているのでここからの出題もない気がします。内容に関しては特に補足することもないので、教科書読んで下さい。

以上、7 章と 9 章以外はある程度説明できていると思います。7 章は正直今からシケプリ化するパワーがありません・・・＞＜