アルゴリズム入門 2010~2015 共通問題　解答解説

- この解答解説は 2016 年度アルゴリズム入門の試験対策のために、教科書を参考にしながら掲載されている 2010~2015 年度の共通問題のシケプリをまとめて適宜編集、加筆したものである。各年度の教科書を除く参考出典は以下の通り。
- この解答解説は筆者の勝手な解釈に基づいており、その正しさは保証しかねます。内容を吟味し正誤を見極めどう活かすかはあなた次第です。
- 解説中のコードには各関数が返す値に return が付いているが、これは筆者の勝手な流儀であり、もちろん無くても良い。また、配列の長さを取得するのに教科書通りの.length()ではなく.length と記述してあるが、これもどちらでも構わない。特にコードには解答例以外にも正答は存在するのでそこは各自の判断に任せる。
- 解説中のコードでは特別な意味を持つ予約語を色で強調してある。

2016 年 11 月　理 1　某クラス　シケ対　Seo

年度	問題番号	著者	参考出典名
2015	――	――	――
2014	問題３	touyou(@touyoubuntu)	アルゴリズム入門 2013-2014 過去問解答*[1]
2013	問題１	touyou(@touyoubuntu)	アルゴリズム入門 2013-2014 過去問解答
2012	問題２	codereading.com	ソースコード探検隊/分布数え上げソート *[2]
2011	問題３	イショティハドゥス 不詳	informatical reality -2013 winter-*[3] 疑似乱数-Wikipedia*[4]
2010	問題３	イショティハドゥス 暇人	informatical reality -2013 winter- 情報科学 2010 年度 冬学期 解答＆解説*[5]

これら素晴らしいシケプリを制作してくださった先輩方に感謝申し上げます。

[1] https://todai.info/sikepuri/search/down.php?id=1754
[2] http://www.codereading.com/algo_and_ds/algo/counting_sort.html
Copyright 2007-2014 codereading.com
[3] https://todai.info/sikepuri/search/down.php?id=1673
[4] https://ja.wikipedia.org/wiki/%E6%93%AC%E4%BC%BC%E4%B9%B1%E6%95%B0
[5] https://todai.info/sikepuri/search/down.php?id=944

目次

2015年度　共通問題

問題1

トリボナッチ数列を題材としたアルゴリズム構築の問題。最後にはアルゴリズムの計算量に関連した問もある。３番目に行列を用いたアルゴリズムが突如登場して驚くかもしれないが、線形代数で学んだように行列はある線形写像に対応しており、線形写像とは極端に言えば実数倍して足し引きすることに他ならないので結局やることは２番目と何も変わらない。ただ２番目でアレヤコレヤ足し引きしていたのを行列の演算で表せますか？という問

解答例

(1) (A) `n < 2`　(B) `n == 2`　(C)　`tri_rec(n-1) + tri_rec(n-2) + tri_rec(n-3)`
(2) (D) `p1`　(E) `c`　(F) `p1 + p2 + p3`
(3) (G) `j == 0 || j == i+1`　(H) `n - 2`
(4) 関数 `tri_rec(n)`の計算量のオーダーは $O(3^n)$であり、
　　関数 `tri_loop(n), tri_mat(n)`は $O(n)$であるから、
　　計算量の観点からは後者２つのほうが優れている

解説

(1)
トリボナッチ数列の初期値と漸化式を素直に入れましょう
３つ前までの値を求めるために自身の関数を呼んでおり、再帰の形になる
つまり、目的の最終点から遡っていくわけで単純だが計算量が膨れ上がる原因になる

```
def tri_rec(n)
  if n < 2
    return 0
  else
    if n == 2
      return 1
    else
      return tri_rec(n-1) + tri_rec(n-2) + tri_rec(n-3)
    end
  end
end
```

(2)

先とは対照的に初めの方から順番に計画的に求めていく

ここで p1, p2, p3 が漸化式における３つ前までの項の値に対応しているのに気付こう

c が新たに求まった項の値T_iであり、次の漸化式の p1 になる、つまり

反復部分一回が漸化式一回分であり、そしたら次の為に p1, p2, p3 を一つずらす

```
def tri_loop(n)
  p3 = 0
  p2 = 0
  p1 = 1
  c = 1
  for i in 4..n
    p3 = p2
    p2 = p1
    p1 = c
    c = p1 + p2 + p3
  end
  return c
end
```

(3)

行列を用いて(2)と同じことをします

初めに行列に対応する配列を用意して、二重ループ内で配列の要素を初期化し

最後にこの行列の演算をするただ、n×m の二次元配列のどちらが行、列に対応しているか釈然としないがここでは n×m の行列に対応するとする

(ただし、たとえ逆だとしても累乗で行列を左側から掛けると解釈すれば行基本変形を施すことになり何の問題もない)
ところで、問題中のアルゴリズムの最後に返している b[0][0]が求める値らしい
そして、ここで思い出して欲しいのは、行列を掛けるとはもとの行列に何か線形的な操作をすることであり、matpower(a,n)関数で行列 a を n 乗するとは、つまり a に行列 a が表す何らかの(n-1)回の反復操作を意味する
そこで演算の過程での 3×3 行列の成分を

$$\begin{matrix} T_n & T_{n-1} & T_{n-2} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{matrix}$$ （T_nはトリボナッチ数列 n 項、a_{ij}は何らかの整数）

であると試してみるのが自然だろう（配列の index に揃えて行、列番号も 0 始まり）
これにある行列をさらに掛けて

$$\begin{matrix} T_n + T_{n-1} + T_{n-2} & T_n & T_{n-1} \\ a'_{10} & a'_{11} & a'_{12} \\ a'_{20} & a'_{21} & a'_{22} \end{matrix}$$ つまり $$\begin{matrix} T_{n+1} & T_n & T_{n-1} \\ a'_{10} & a'_{11} & a'_{12} \\ a'_{20} & a'_{21} & a'_{22} \end{matrix}$$

となれば漸化式っぽくなり嬉しいのになあ、と次に思う
上二つを比べて気づかなくてはならないのは、列(基本)変形をすればいいじゃん発想
0 列目に 1,2 列目を足して、1 列目に 0 列目を足して自身を引き、
2 列目に 1 列を足して自身をひく
この変形に対応する変形行列は、単位行列

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

に同じ変形を施した

$$\begin{matrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{matrix}$$

に他ならない（詳しくは線形代数の教科書でも参照のこと）
この時、0 行目成分がT_3, T_2, T_1になるのに気づく
よって、この行列を n 乗すると左上にT_{n+2}が現れる
（しつこいようだが、列に関して漸化式に対応した変形を(n-1)回施したことになる）
これを Ruby でかくと

```
def tri_mat(n)
  a = make2d(3,3)
  for i in 0..2
    for j in 0..2
      if j == 0 || j == i+1
        a[i][j] = 1
      else
        a[i][j] = 0
      end
    end
  end
  b = matpower(a,n-2)
  return b[0][0]
end
```

＊補足

```
  a = make2d(3,3)
```

の関数の中身は

```
  a = Array.new(3)
  for i in 0..2
      a[i] = Array.new(3)
  end
```

これと同じ

＊補足

予め用意された関数 matpower()が気になるならば、考えてみよう

まず、与えられた二つの行列の積を対応する二次元配列で返す関数 matmul()はどうなるかというと、

```
  def matmul(a,b)
    if a[0].length != b.length
      return nil
    end
    c = make2d(a.length,b[0].length)
    for i in 0..(c.length-1)
      for j in 0..(c[0].length-1)
        c[i][j] = 0
```

```
      for k in 0..(a[0].length-1)
        c[i][j] = c[i][j] + a[i][k]*b[j][k]
      end
    end
  end
  return c
end
```

a×b を求めたいのだが、初めに一応、積が定義できるか確認してから
　出来上がる行列に当たる配列 c を用意して、要素を埋めていく
matpower() は matmul()を繰り返し呼び出して

```
def matpower(a,t)
  if a.length != a[0].length
    return nil
  end
  b = make2d(a.length,a.length)
  for i in 0..(b.length-1)
    for j in 0..(b.length-1)
      b[i][j] = a[i][j]
    end
  end
  for i in 2..t
    b = matmul(b,a)
  end
  return b
end
```

そもそも正方行列でないと累乗できないから確認して
新たに配列 b を用意して初めに a をコピーしてから a を(n-1)回掛ける

(4)

再帰的な場合は一つの関数から呼び出される自身の関数の数を基底とした指数になり、一重のループ処理を含む場合はその反復回数になる

計算量のオーダーに関して簡単に説明する
計算量とはアルゴリズムの実行に要する時間の指標であり、オーダーがその大まかな尺度と言える。オーダーが n ならば実行時間は n の大きさに比例すると解釈する

なぜ、そんなに時間を気にするかって？実際の計算はここで扱うのとは比較にならないほど膨大で複雑な計算をするのだが、当然実行から解を得るまでに長い時間がかかる。すると、明日の天気を超高確率で予測可能な計算があるが、計算に一週間かかります、などと本末転倒な事態にもなりかねない。計算に要する時間は大きな関心なのだ。

(1)のアルゴリズムでは一回目に関数 tri_rec(n)を呼ぶと三回また自身を呼ぶので、引数が n, n-1, n-2, と減ってゆき最後２以下になるまで続く。よってオーダーは3^nであり、O(3^n)と表す。何？n は 1 まで減らないから n 乗じゃないって？計算量のオーダーは巨大な計算つまり n が十分大きいことを想定しており、細かい端数の影響は無視し計算に要する時間が大体3^nに比例することが重要なのである。再帰的な場合はこのように一つの関数から呼び出される自身の関数の数を基底とした指数になるだろう。また、3^nは n の増加に対してすぐに途轍もなく大きな値に発散するから論外だと分かり、O(n)の関数 tri_loop(n), tri_mat(n)の方が優れている。

問題2

(1)
計算機では無限の精度は表現できず二進無限小数は近似的にしか表されない
という現実的制約から誤差が発生することに尽きる
いわゆる浮動小数点誤差とか丸め誤差というやつである
さらに計算の過程でこの誤差がいろいろ悪さしてさまざまな誤差の原因となる

解答例

1.
値 a,b,c は二進数表現では無限小数となり浮動小数点誤差を生じる可能性があるから。
2.
b^2と 4ac の大きさに対してその差は極端に小さいため減算すると有効桁数が減り、桁落ち誤差が生じるから。
3.
4ac の値はb^2に対して極端に小さいため、加算すると 4ac に由来する部分の桁が浮動小数の有効桁数を超えて、情報落ち誤差が生じるから。

解説

1.

計算機の中では数値を二進数で扱うのは周知として、
なおかつ計算機のリソースは有限のため無限の精度は表現できない
つまり、二進数での無限小数は近似的にしか表現できない
(Ruby の浮動小数は多くの場合 64bit の指数の形で小数を表現しており、有効桁数は環境にこそ依れ、せいぜい 15 桁程度とのこと)
そのため、浮動小数の計算ではこれに由来してどうしても誤差が発生してしまう。
計算の結果の有効桁数外側に誤差が収まっていれば問題ないのだが、この問題のように計算の過程で誤差が強調されてしまい、実際に信用できる桁数が落ちることがある。
だから誤差が計算過程でデカくなり表に出てこないように気をつけよう

2.

b^2-4ac のように極端に 0 に近い値になる加減算を行うと、有効桁数が減ってしまう。
計算機は浮動小数の有効桁数を保とうとして失われた桁数を 0 で埋めるため、誤差が上の桁に上がってきてしまう。
(イメージ)　有効桁数 8 桁で計算すると、

1.2345678 − 1.2345666 = 0.0000012　有効桁数が 2 桁まで落ちる

3,

b^2はとても大きな値である一方 ac の値がとても小さいため
ac 由来の桁の部分が有効な桁に入らず浮動小数点誤差に紛れて捨てられてしまい
(-b+b)/2a という感じになった
(イメージ)　有効桁数 8 桁で計算すると、

1234567.8 + 0.0012345678 = 1234567.8012345678

有効桁数 8 桁までに変化なし

誤差まとめ

浮動小数に伴う誤差を簡単にまとめると、

・丸め誤差（浮動小数点誤差）

二進無限小数になる数値は小さな桁の部分を丸めて近似的に表現するために発生。
有効桁数は倍精度で大体 15 桁　ただし、2^{-n}のような二進有限小数でかつ有効桁数内で表せる値は厳密に表現できることもある。

・桁落ち誤差
値が極端に近い二数を減算すると、有効桁数が大きく減少する
・情報落ち誤差
ある値に対し極端に小さい値を加減算しても、有効桁数の外側になってしまい、丸め誤差に紛れて捨てられてしまう
・打ち切り誤差
関数を多項式で表現できるテイラー展開など、たとえ望むだけの精度が得られるとしても、計算機のリソースなど現実的な制約のため途中で打ち切ることによって生じる誤差。

などが試験にも頻出の模様で要注意

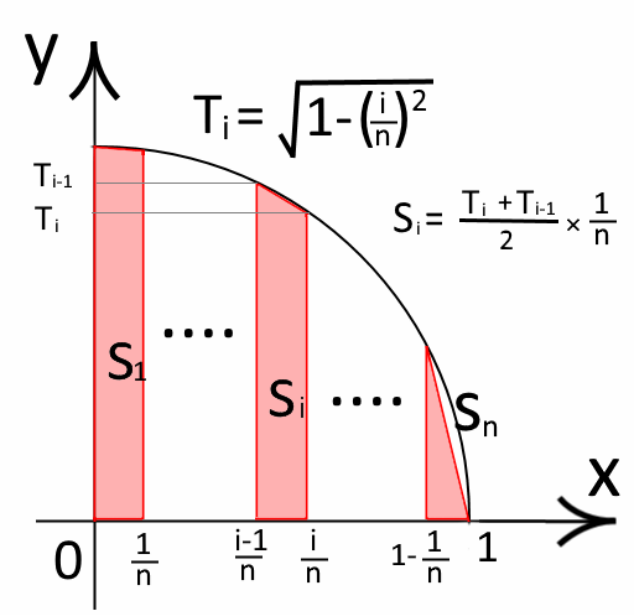
(2)

解答例

```
1.
 (A) delta * i  (B) s * delta
2.
台形公式で近似的に四分円の面積を求めているため、その値は真の値と異なるから
円周率の真の値とは異なる値を得る
また、この近似値と真の値の差は n が大きい程小さくなるから
得た値は n が大きくなると円周率に近づく
3.
No
n が十分大きくなると丸め誤差の蓄積による誤差の影響が大きくなり、また n が極端に大きくなると s の加算時に情報落ち誤差が発生して大きな誤差になり得るから。
```

解説

1.
問題中に与えられたアルゴリズムの断片だけ眺めていても分かりづらい
そこで、台形公式で円の面積を求めるらしいので
素直に自分でアルゴリズムを考えてみる



図のように台形に円を分割して面積を近似的に求める
delta は文字からも n 分割した、台形の高さと想定できて delta = $\frac{1}{n}$
i 番目の台形は(i = 1....n)
右下の点が(delta×i, 0)
右上の点が($\sqrt{1 - \text{delta} \times \text{i}}$, 0)
よってこの台形の面積は
$\frac{\sqrt{1-\text{delta}\times\text{i}}+\sqrt{1-\text{delta}\times(\text{i}-1)}}{2}\times$delta
この流れをそのまま形にすれば

```
def pi(n)
  delta = 1.0/n
  s = 0.0
  for i in 1..n
    s = s + (sqrt(1-(delta*(i-1))**2)+sqrt(1-(delta*i)**2))*delta/2.0
  end
  return s * 4
end
```

ん、なんか違うぞ。解答はもっとスッキリしているなあ。なので、これから今のソースを整理してみると、まず、台形の面積を求めるとき毎回 delta を掛けずにそのまま足して、最後に和に掛ければいいじゃん！反復部分を以下のように改造

```
・・・・
  for i in 1..n
  s = s + (sqrt(1-(delta*(i-1))**2) + sqrt(1-(delta*i)**2))/2.0
  end
  return s * delta * 4
end
```

次に、i 番目の台形の右底辺と(i+1)番目の左底辺は同じで
今のままでは二回同じ計算をしている
なら一回で済ますには足すときに÷２しなければいい
ただし、1 番目の左底辺と n 番目右だけは一回しか登場しないから 2 で割る

```
def pi(n)
  delta = 1.0/n
  s = 0.0
  for i in 0..n
    if i == 0 || i == n
      s = s + sqrt(1-(delta*i)**2)/2.0
    else
      s = s + sqrt(1-(delta*i)**2)
    end
  end
  return s * delta * 4
end
```

ただ、1 番目の左底辺は 0.5、n 番目右は 0 と分かっているから
これを反復部分から除いて完成

```
include(Math)
def pi(n)
  delta = 1.0/n
  s = 1.0/2.0
  for i in 1..(n-1)
    s = s + sqrt(1-(delta*i)**2)
  end
  return s * delta * 4
end
```

ただし、s に初期値として 1 番目の左底辺 0.5 が入る

3.

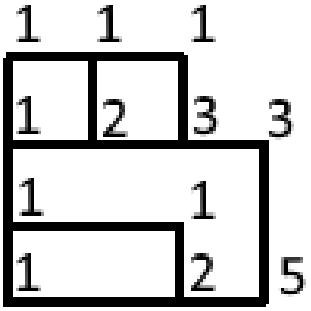
アルゴリズムの論理からすれば n の増加に対して精度は上がり続けるのだが、計算機で表現する限り浮動小数の有効桁数という現実的な制約があるというお話し。
特に n が極端に大きくなっていくと、1 つ 1 つの台形が小さくなり、s に加算するときそれまで足してきた s に対して極端に小さくなるため、結果情報落して以降は足しても s が変化せず変な値になる。

問題3

解答例

(1) 5 通り
(2)　(A) WE[0,x-1]　(B) NS[y-1,0]　(C) WE[y,x-1]　(D) NS[y-1,x]
(3)　(E) return 1　(F) return cp(0,x-1) * WE[0,x-1]
　(G) return cp1(y-1,0) * NS[y-1,0]
　(H) return cp1(y,x-1) * WE[y,x-1] + cp1(y-1,x) * NS[y-1,x]
　　＊もちろん return を省いてよい
(4)　(I) 0..y　(J) 0..x　(K) return 1
　　(L) return count[0][x-1] * WE[0,x-1]
　　(M) return count[y-1][0] * NS[y-1,0]
　　(N) return count[y][x-1] * WE[y,x-1] + count[y-1][x] * NS[y-1,x]
(5)　xy
　関数 calc()の計算量は cp2(y,x)の引数に依存せず、
　関数 cp2()内に二重の反復処理があり、
　それぞれ反復回数が y, x に比例するから

解説



(1)　数えましょう　　図参照のこと

(2)

具体的に考えよう。もしすべての経路が存在するならば、地点(y,x)に来る道は北か西のどちらかであり、

S(y,x) = S(y,x-1)×1 + S(y-1,x)×1

地点(y,x)に北からくる道がなければ

S(y,x) = S(y,x-1)×1 + S(y-1,x)×0

西からくる道がないと

S(y,x) = S(y,x-1)×0 + S(y-1,x)×1

ここで、北からの道の有無は NS[y-1,x]の 1,0 に
西からの道は WE[y,x-1]にそれぞれ対応していることに気づけば終わりで

S(y,x) = S(y,x-1)×WE[y,x-1] + S(y-1,x)×NS[y-1,x]

と一般的に表され、あと x=0,y=0 の場合を個別に対応するだけ

結果次のようになる。

```
S(0,0) = 1
S(0,x) = S(0,x-1)×WE[0,x-1]
S(y,0) = S(y-1,0)×NS[y-1,0]
S(y,x) = S(y,x-1)×WE[y,x-1] + S(y-1,x)×NS[y-1,x]
```

(3)

```
def cp1(y,x)
  if y == 0
    if x == 0
      return 1
    else
      return cp1(0,x-1) * WE[0,x-1]
    end
  else
    if x == 0
      return cp1(y-1,0) * NS[y-1,0]
    else
      return cp1(y,x-1) * WE[y,x-1] + cp1(y-1,x) * NS[y-1,x]
    end
  end
end
```

(2)の漸化式を形にするだけ
関数内側から自身の関数を呼んでいる
漸化式の S()を cp1()に代えると思えば分かりやすい

(4)

```
def cp2(y,x)
  count = make2d(y+1, x+1)
  for i in 0..y
    for j in 0..x
      count[i][j] = calc(i, j, count)
    end
  end
  return count[y][x]
end
```

```
def calc(y, x, count)
  if y == 0
    if x == 0
      return 1
    else
      return count[0][x-1] * WE[0,x-1]
    end
  else
    if x == 0
      return count[y-1][0] * NS[y-1,0]
    else
      return count[y][x-1] * WE[y,x-1] + count[y-1][x] * NS[y-1,x]
    end
  end
end
```

計算に必要なのは地点(i, j) (0≤i≤y, 0≤j≤x)での S(i, j)
そこで初めにこの(y+1)×(x+1)個の S(i, j)に対応する変数として
二次元配列 couunt[][]を用意する
あとは counu を埋めていくだけ
このとき値が関数 calc()から返されるのだが
その中身は cp1()とほぼ同じ
再帰関数とは違い cp1()と自身を呼ぶかわりに count[][]を参照している
(今回は cp2()と calc()の変数 count は同じものを指している)

(5) 解答例の通り
計算量のオーダーに関して詳しくは問題１(4)や教科書を参照

2014 年度　共通問題

問題1

解答例

```
(1)  (A) a.length == 0   (B) 1..(a.length-1)   (C)1..(b.length-1)
     (D) a[0] + b[0]
(2)  (E) a.length == 0   (F) inner(v, a[0])     (G) 1..(a.length-1)
(3)  (H) a.length == 0
     (I) vmmult(a[0], b.transpose) もしくは vmmult1(a[0], b)
     (J) 1..(a.length-1)
```

解説

＊注意

a[k]で参照される配列は行列 a の k 行目の行ベクトルに対応している。しかし、計算で欲しい列ベクトルに対応した配列は一発では得られない。そこで a.transpose と行列 a の転置行列を求めたわけで、すると、転置行列 a.taranpoose の行番号 i を a.tranpose[i]と指定すると元の行列 a の i 列の列ベクトルが返るから便利。今回は何も転置を求めること自体には意味はない。転置行列の性質とか必死に思い出さないように

(1)

ベクトルの内積を求めたいのだから、対応した配列の同じインデックスの値どうしの積を足していけばいい。これを再帰的に表現するには、配列の頭だけに注目して、もし配列が空ならば当然 0 を返して、配列が空でなければ頭の a[0]と b[0]の積だけ計算し残りの計算は自身の関数をまた呼ぶことで行う。この時、引数の配列はさっき計算した先頭を除く a[1..(a.length-1)]と b[1..(b.length-1)]になる。（配列 a の最後の要素のインデックスは a.length でなく a.length-1）

すると、関数 inner()はこんな感じになる

```
def inner(a, b)
  if a.length == 0
    return 0
  else
    return a[0] * b[0] + inner(a[1..(a.length-1)], b[1..(b.length-1)])
  end
end
```

次に和を求めたいのだが、今度はベクトルが出てくるので対応する配列で返すのに気を付けて、あとは同様に配列の先頭にだけ注目して足していけばいい。

```
def add(a, b)
  if a.length == 0
    return []
  else
    return [a[0] + b[0]] + add(a[1..(a.length-1)], b[1..(b.length-1)])
  end
end
```

(2)

ベクトルを行列の左側から掛けると解釈して、ベクトル v と行列 a の各列ベクトルの内積を配列に順に詰めていって返せばいい。ここで a.tranpose[i]と参照すると元の行列 a の i 列の列ベクトルが返るから、後は同様に配列の先頭に注目して

(関数 vmmul1()内の行列 a は vmmul()内の行列 a の転置であり、同じではない)

```
def vmmult1(v, a)
  if a.lnegth == 0
    return []
  else
    return [inner(v,a[0])] + vmmul1(v, a[1..(a.length-1)])
  end
end
```

(3)

行列 a を行列 b の左側から掛けると解釈して、行列 a の各行ベクトルと行列 b の積（求めたい行列の行ベクトルに対応した配列で返される）を順に配列に詰めて二次元配列で返す。これを再帰的に表現すると、配列 a の先頭に着目して、先頭の要素つまり行列 a の行ベクトルと行列 b の積だけ計算して残りはまた自身の関数を呼んで行う。

＊注意
mmmult1()内の行列 b は mmmult()に渡した元の行列 b の転置である。行列 a の各行ベクトルと元の行列 b の積を求めるとき、行ベクトルを左側から掛けて元の行列 b の各列ベクトルとの内積を求めて欲しいわけで、vmmult(a[0],b)だと元の行列 b の各行ベクトルとの内積を取ってしまい結果的に掛ける方向が逆になる。だから vmmult(a[0],b.transpose)か vmmult1(a[0],b)とする。そもそも、こうも回りくどくなったのは mmmult()から呼ばれたとき b が既に転置になっているからであり、この転置の意図は解せない。(意地悪なのか？)
やるなら mmmult(),mmult1()の二つなんて用意せずに

```
def mmmult(a, b)
  if a.length == 0
    return []
  else
    return [vmmult(a[0],b)] + mmmult(a[1..(a.length-1)],b)
  end
end
```

だけでいいじゃないか！

問題2

解答例

(1) P(t,a)= $\begin{cases} q * P(t-1, a+1) & (a < 2) \\ p * P(t-1, a-1) & (a > 8) \\ p * P(t-1, a-1) + q * P(t-1, a+1) & (1 < a < 9) \end{cases}$

(2) アルゴリズムが再帰的であるため、計算量のオーダーが $O(2^t)$ となり
小さな値でも計算量が膨れ上がるから

(3) (A) 0 (B) q * prob[i-1][j+1] (C) p * prob[i-1][j-1]
(D) p * prob[i-1][j-1] + q * prob[i-1][j+1] (E) return prob[t][a]

(4) n*t

(5) 計算量の指標としておおまかに実行に要する時間を見積もることができ、
これによりアルゴリズムの性能を評価したり、実行に膨大な時間を要する
アルゴリズムを事前に見つけ出すこともできる。

解説

(1)

t 回目に a 枚所持しているのだから、(t-1)回目には(a+1)枚もしくは(a-1)枚持っていいたことになる。ただし a の値によって場合分けが発生し、これはコード中の条件分岐に対応している。

(2)

再帰的なアルゴリズムとはある関数のなかに自身の関数が含まれているものを指す。ただし、関数の引数は簡約化されていくので循環はせずいつかは終わる。今回のアルゴリズムを再帰的に表現すれば、

```
def P(t,a)
  if t == 0
    if a == 5
      return 1
    else
      return 0
    end
  else
    if a < 2
      return q * prob[i-1][j+1]
    else
      if a > 8
        return p * prob[i-1][j-1]
      else
        return p * prob[i-1][j-1] + q * prob[i-1][j+1]
      end
    end
  end
end
```

1 つの関数内から自身を 2 回呼ぶから計算のオーダーは $O(2^t)$ となる。

a の値によっては 2 回も呼ばれないじゃん、と言いたくなるかもしれないが、オーダーが問題となるような膨大な計算量を普通は想定するから、t は十分大きく、a の値によってどうこうの影響は無視して計算量がおよそ2^tに比例することのみ注目する。

(3)

今度は t=0 から計画的に求める。

初めに各場合の確率を格納する二次元配列 prob[][]を用意してから prob[0][]つまり初期状態を設定する。あとは漸化式に従って順番に prob[][]を埋めていくだけ。

最後に求めるべき確率 prob[t][a]を返す。

```
def P_loop(t,a)
  prob = make2d(t+1,11)
  for j in 0..10
    if j == 5
      prob[0][j] = 1
    else
      prob[0][j] = 0
    end
  end
  for i in 1..t
    for j in 0..10
      if j == 0 || j == 1
        prob[i][j] = q * prob[i-1][j+1]
      else
        if j == 9 || j == 10
          prob[i][j] = p * prob[i-1][j-1]
        else
          prob[i][j] = p * prob[i-1][j-1] + q * prob[i-1][j+1]
        end
      end
    end
  end
  return prob[t][a]
end
```

(4)

t が十分大きい場合を想定するから、始めの初期化のループは無視して、prob[][]を求める二重ループ処理の反復回数がそれぞれ t,n に比例するから O(t*n)

問題3

解答例

(1)整数型の数値どうしの徐算は商を返すから。
(2)値の大きさが近い 1 と 1+1/a を減算すると桁落ち誤差が発生するから
(3)a^{-5}は 1 に対してその丸め誤差と同程度に小さいため 1+a^{-5}と 1 を減算するとさらに大きな桁落ち誤差を生じる可能性があるから。
(4) a^{-6}は 1 に対し極端に小さいため 1+a^{-6}と 1 の減算で情報落ち誤差が発生したから。
(5)1.0　1/1024.0=2^{-10}のような 2 進有限小数で表される値は厳密に計算機内で表現されるから
(6)0.0　a^{-6}は 1 に対して極端に小さいため情報落ち誤差が発生したから。
(7)3.0
(8)4-π/4

解説

(1)

Ruby を始め多くの言語では同じ数値でも整数型と浮動小数型で計算機内での表現が異なり演算時の扱いも違う。a=1000.0 とすれば a は浮動小数型だが a=1000 では整数型と認識される。そして整数型どうしの除算は商が得られるため 1/a=0 になってしまう。これを回避するには割る数と割られる数の少なくとも一方を浮動小数型にすれば、演算時に高い精度の型に合わせて浮動小数として計算され 1/a=0.001 になる。

(2)(3)

どちらも桁落ち誤差と言えるが、大きさ 1 に対して 1+a^{-5}と 1 の差は 1 と 1+1/a の差よりも一層小さく、浮動小数点誤差と同程度まで小さいため、有効桁数の減少が激しいと解釈できる。言ってみれば桁落ち誤差と情報落ち誤差の狭間であり、もし丸め誤差と本当に同程度の大きさならば 1+a^{-5}と 1 の減算で得られる値はもはや誤差に由来する何かでしかない。

(4)

浮動小数点の有効桁数は環境にこそ依れ、10 進表現で高々15 桁程度のようなので、a^{-6}は 1 に足したら明らかに誤差の彼方である。

(5)(6)

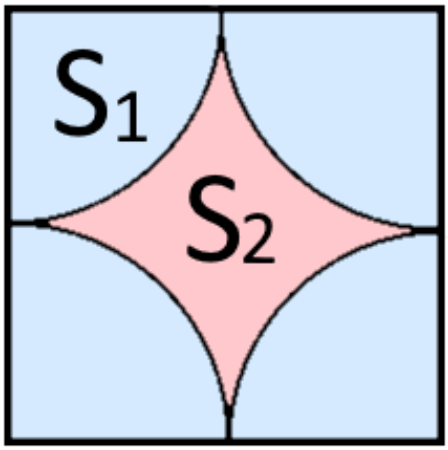
通常 10 進小数は二進表現で無限小数となり近似的にしか表せないが、解答の通り二進有限小数となる限られた場合のみ厳密に表現される。もっとも、情報落ちしたら話は別である。

(7)(8)

モンテカルロ法の数値積分の一例である。すなわち、ランダムに点をプロットしていき点がある図形内に含まれた回数を数え点の総数で割る。この値は点の総数を増やせばその図形にランダムな点が含まれる確率に収束し、その確率は全領域と図形の面積の比率に比例するというやつである。

コードを眺めてみると、どうやら{(x,y)|0≤x<1, 0≤y<1}の正方形の各頂点からの距離が 0.5 を超えれば 1 足すようなので図のS_1部分に点があれば 3 足してS_2部分に点があれば 4 足す。よって点の総数 n が十分に大きければ全体の正方形の面積に対する面積 $3S_1+4S_2$の比率に収束する。この正方形は面積 1 なので、求める値はそのまま面積 $3S_1+4S_2$で

$S_1=\pi/4$, $S_2=1-\pi/4$　より　$4-\pi/4$



問題4

解答例

(1) (A) C[i][0] (B) (i+1) (C) C[j][1] (D) return max
(2) N^3
(3) (E) rcvr[n-1][m-1] + C[n-1][m-1]
(4)

```
def g()
  revr = make2d(N+1,M+1)
  for i in 0..N
    for j in 0..M
      if i==0 || j==0 || i<j
        revr[i][j] = 0
      else
        max = rcvr[i-1][j-1] + C[i-1][j-1]
        if max < rcvr[i-1][j]
          max = rcvr[i-1][j]
        end
        rcvr[i][j] = max
      end
    end
  end
  return rcvr[N][M]
end
```

解説

(1)
盗賊と捜査官の可能な全組み合わせを虱潰しに調べる。そして初めの捜査官の回収金額 rcvr_i と次の捜査官の rcvr_j の和に当たる総回収金額 rcvr の最大値を求める。
(2)
(1)からも分かる通り捜査官の人数 M だけ反復処理が必要でその反復回数は大体 N だから計算量のオーダーはN^M

(3)

分かりやすさのため、以下捜査官と盗賊が何番目か数えるとき普通に 1 始まりとする。n 人の盗賊に m 人の捜査官を配置したときの最大回収金額 rcvr[n][m]を知ろうとする時、rcvr[i][j](0≤i<n,0≤j≤m,j≤i)の値は既知である、言い換えれば(n-1)番目までの盗賊に k 人の捜査官をどう配置するのが最善か k=m,m-1,m-2…の全ての場合で把握している。すると問題になるのは n 番目の盗賊に捜査官を配置するか否かである。

もし配置するならば(n-1)番目までの盗賊に(m-1)人の捜査官を配置するから、

rcvr[n][m] = rcvr[n-1][m-1] + C[n-1][m-1]

配置しないなら(n-1)番目までの盗賊に m 人の捜査官を配置するから、

rcvr[n][m] = rcvr[n-1][m]

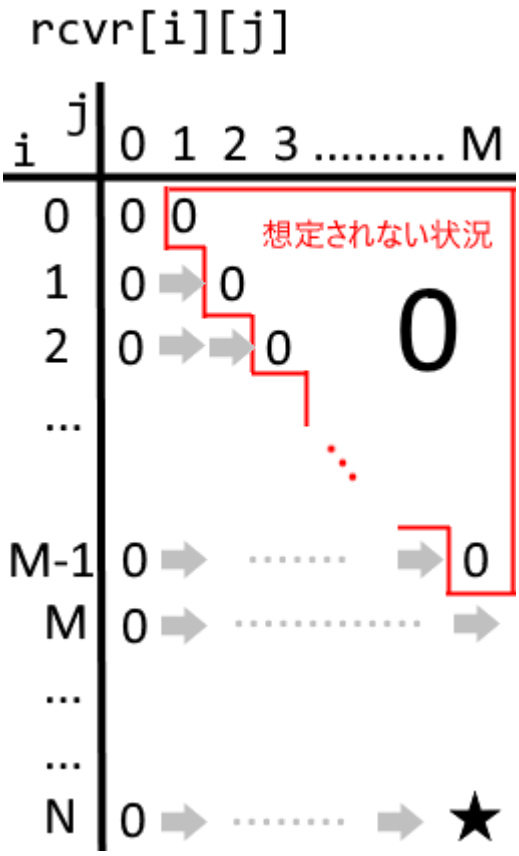
この二つの値のうち大きい方を選べばいい。

ただし、捜査官の士気がどうこうで 0≤m≤n の制約があるため、m=0,n=0,n<m ならば

rcvr[n][m] = 0

(4)

はじめに各場合の最大回収額を格納する二次元配列 rcvr[][]を用意して、後は(3)の漸化式に従って順に rcvr[][]を埋めていく。図のイメージ



＊補足

二次元配列を用意する関数 make2d()は配布コードを実際に眺めればわかるのだが、配列を宣言して各値を数字 0 で初期化してある。この豆知識を使うと、解答例の面倒な条件分岐が減らせて、

```
def g()
  rcvr = make2d(N+1,M+1)
  for i in 1..N
    for j in 1..M
      max = rcvr[i-1][j-1] + C[i-1][j-1]
      if max < rcvr[i-1][j]
        max = rcvr[i-1][j]
      end
      rcvr[i][j] = max
    end
  end
  return rcvr[N][M]
end
```

この時反復処理内のインデックスが 1 始まりだということに気を付けよう。うっかり 0 始まりにすると C[-1][-1]なる参照をしてしまう。これは Ruby だとエラーにはならず C[c.length-1][C[C.lemgth-1].length-1]つまり後ろから 1 番目を参照してしまうのでエラるよりも猶更厄介である。ただ問題なのが、配列 C[i][j]に対して問題の設定上想定されない i<j の場合も参照していいのか疑問ではある。0 だったらいいが、なにも初期化されておらず nil だったら困るので j≤i の制約は残したほうがよさそう。そもそも、関数 make2d()が授業中のと同じとは明記されておらず、本当に 0 に初期化された二次元配列なのかは判断できない。結論としては、面倒でも無難に解答例通りいきましょう。

2013 年度　共通問題

問題1

解答例

```
(a) (ア) from[n-1]  (イ) (n+1) / 2 (ウ) return from[0]
(b)  n  from[][]
     8  [[7],[2],[8],[3],[1],[4],[6],[5]]
     4  [[2,7],[3,8],[1,4],[5,6]]
     2  [[2,3,7,8],[1,4,5,6]]
     1  [[1,2,3,4,5,6,7,8]]
(c)
```

単純整列法の時間計算量のオーダーは $O(n^2)$、空間計算量は O(1)であり、併合整列法はそれぞれ O(nlogn)、O(n)であるから、実行時間の観点からは後者が優れているが使用メモリの観点からは前者の方が優れている。

解説

(a)(b)

(b)のように具体的に手を動かした方が分かりやすいかもしれない。

二つの数字の山を混ぜて要素を小さい順に並べひとつの山にする併合の作業を繰り返し、最終的に全体をひとつにする。このとき混ぜていく数字の山は from[]に格納されている内側の配列に当たり、併合した山は配列 to[]に詰めて最後に from と入れ替わる。

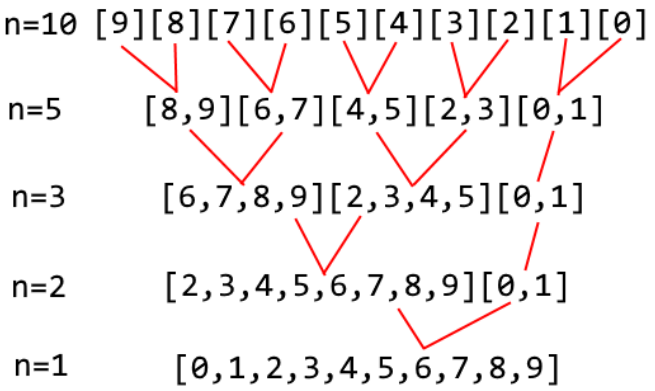
最初は各山に数字ひとつずつ詰め込む。そしたら関数 merge()を用いて隣り合う山と山を併合していきたい。ここで、山の数 n が偶数ならばきれいに二組を n/2 個つくれるが、n が奇数だと最後の山(インデックスは n-1(ア))があぶれるため個別対応が必要で、これが if !is_even(n)つまり if n%2 == 1..の部分に当たり、あぶれた山は何もせずそのまま詰め込み計(n+1)/2 個の山ができる。整数どうしの除算は商になるのに注意して結局新しい山は(n+1)/2 個((イ),to = make1d((n+1)/2)に該当し偶奇の場合分けは要らない)。最後には一つの山になり from[0]に配列として格納されているのを返して終わり。

(b)のようにnが奇数になる場合も具体的に書き出してみると、

a = [9,8,7,6,5,4,3,2,1,0]

のとき from の中身は

右図のように変化していく



(c)

教科書で扱ったとか言われても困るのでコードを書いてみよう。

単純整列法とは、元の配列の最小の要素を抜き出しさらに残りのうち最小要素を探して、と繰り返していく単純なアルゴリズム。ここでは抜き出した要素は配列の先頭から順に詰めていくことにしよう。すると、

```
def simple_sort(a)
  for i in 0..(a.length-1)
    minIndex = i
    for j in (i+1)..(a.length-1)
      if a[j] < a[minIndex]
        minIndex = j
      end
    end
    v = a[i]
    a[i] = a[minIndex]
    a[minIndex] = v
  end
  return a
end
```

と書ける。このアルゴリズムでは配列 a の長さを n として反復部分が

$\sum_i^{n-1}(n-i-1)$=n(n+1)/2 回繰り返されるので時間計算量は低次の項や定数倍を無視して $O(n^2)$。一方、空間計算量はアルゴリズムの実行に必要な記憶領域メモリの容量の指標であり、今回は大体変数の数が n にどう依存するか注目すればよい。このアルゴリズムでは n への依存性はなく O(1)となる。

次に、関数 mergsort()内の関数 merge()はその仕様を実現する。渡された二つの配列 a,b はソート済みだからインデックスの小さい方からより小さい要素を選んでは新しい配列に詰め込んでいけばよさそう。そして a,b どちらかの配列から全要素抜き出したら、もう一方の配列の残り要素をそのままの順番で詰め込んで完成。

```
def merge(a,b)
  i = 0
  j = 0
  c = []
  while i < a.length && j < b.length
    if a[i] < b[j]
      c.push(a[i])
      i = i + 1
    else
      c.push(b[j])
      j = j + 1
    end
  end
  while i < a.length
    c.push(a[i])
    i = i + 1
  end
  while j < b.length
    c.push(b[j])
    j = j + 1
  end
  return c
end
```

関数 mergsort()と合わせてみよう。初めの数字の山の数を n=2^m とすれば、mergsort()内の山二つを併合する反復部分は m=$\log_2 n$回繰り返され、その度に呼ば

れる関数 merge()では n 回の反復があるから大体計 $n\log_2 n = n\log n \times \log_2 e$ 回の繰り返しがある。よって定数倍を無視して時間計算量のオーダーは O(n log n)。使った変数では、配列 from[][]の大きさが n×1 であるから空間計算量は O(n)。
解答例のように、一般にアルゴリズムの時間的・空間的計算量は背反の関係にある。一方が優れていると他方は犠牲になる trade-off な関係である。時間も計算機のリソースも有限なので何を最優先すべきか見極めるのが重要となるだろう。

問題2

解答例

(a)
(i)

i	a[]
1	[1,3,2,5,4]
2	[1,2,3,5,4]
3	[1,2,3,5,4]
4	[1,2,3,4,5]

(ii) （一例） a = [5,4,3,2,1]
並べ替え後 a = [4,3,2,1,5]

(b)
(i) a = [1,2,3,4]
計算量のオーダー$O(2^n)$
(ii) （一例）

```
def min(a,i,j)
  if i == j
    return a[i]
  else
    m = min(a,i,j-1)
    if a[j] < m
      return a[j]
    else
      return m
    end
  end
end
```

計算量のオーダーは O(n)になる

解説

(a)

x 君が実装を試みているのはバブルソートとか隣接交換法と呼ばれる整列法で、配列の隣接する要素の順序が逆ならば入れ替える操作を先頭から(配列の長さ-1)回繰り返す。これを入れ替えがなくなるまで繰り返す。問題のコードはこの二回目の繰り返しが抜けているから失敗。x 君のものをそのまま改良すれば

```
def x_sort(a)
  isEnd = false
  while !isEnd
    isEnd = true
    for i in 0..(a.length-2)
      if a[i] > a[i+1]
        s = a[i]
        a[i] = a[i+1]
        a[i+1] = s
        isEnd =false
      end
    end
  end
  return a
end
```

この時、大きな値は前にあってもすぐに後ろへ移動するが、小さな値が後ろにあると一回のソートでは1しか前にいかなので時間がかかることに気づく。(ii)では一回のソートで整列できないように小さな値を後ろに配置すればよさそう。

＊補足

大小を比べる範囲中の最大の要素は必ず一番後ろに移動するのに注目すれば以下のようにも書ける。計算量のオーダーは配列の長さ n を用いて $O(n^2)$

```
def x_sort2(a)
  for i in (a.lnegth-2)..0
    for j in 0..i
      if a[j] > a[j+1]
        s = a[i]
        a[j] = a[j+1]
        a[j+1] = s
      end
    end
  end
  return a
end
```

(b)

関数内に自身の関数を含み再帰的で、毎回 2 回呼ばれているから大体計2^n回反復する自身の関数を内側から二回も呼ぶからいけないのであって一回で済ますと、計 n 回でおさまる。二分探索法とかもよさそう。

```
  def min(a,i,j)
    if i == j
      return a[i]
    else
    v = min(a,0,i+(j-i)/2)
    c = min(a,i+(j-i)/2,j)
    if c < v
      return c
    else
      return v
    end
  end
```

問題3

解答例

```
(a)
def div (x,y)
  return mul(mul(x,[y[0],-y[1]]), [abs(y)**-2, 0.0])
end
(b)
  前者で10^8に 1 を足す場合は桁落ち誤差が発生する程度だが
  後者で10^16に 1 を足すと情報落ち誤差が発生するから
(c)
def f(x)
  return plus(mul(x,x),plus(x,[1.0, 0.0]))
end
def fs(x)
  return plus(mul([2.0, 0.0],x),[1.0, 0.0])
end
(d)
plus(x,mul([-1.0, 0.0],div(f(d),fd(x))))
```

解説

(a)

全部自分で定義してもいいですが、せっかくなので mul()を流用しましょう。

複素数 y の共役複素数$\bar{y}$は[y[0],-y[1]]であり、

$$\frac{x}{y} = \frac{x\bar{y}}{y\bar{y}} = \frac{mul(x,\bar{y})}{abs(y)**2}$$

大きさは実数なのでそのまま実部虚部の係数を割る。

ちなみに[r,0.0]と x の積を求めれば x の実部虚部係数を r 倍したことと同じ。

(b)

10^8に 1 足してから二乗するか、それぞれ二乗してから足すかの順序がちがう。10^8に対して 1 は十分小さいものの有効桁数には含まれている。一方10^{16}に対し 1 は有効桁数の外側である。結果足したはずの 1 は丸め誤差の彼方に紛れて消えた。

(c)

言われた通り関数を定義しましょう。

(d)

i 回の操作で得られた x の値をx_iとしよう。すると、

接線の方程式は Y-f(x_i)= fd(x_i)×(X-x_i)

Y=0 代入して零点を求めると、

X=x_i-f(x_i)/fd(x_i) これがx_{i+1}になる。

減算は-1 を掛けてから加算で plus(x_i , mul([-1,0], f(x_i)/fd(x_i)))と表現する。

問題4

解答例

```
(a)
def probability(i,j,k)
  if i+j+k == 0
    return 1.0
  end
  prob = 0.0
  if i > 0
    prob = prob + probability(i-1,j,k)*w(i-1,j,k)
  end
  if j > 0
    prob = prob + probability(i,j-1,k)*l(i,j-1,k)
  end
  if k > 0
    prob = prob + probability(i,j,k-1)*d(i,j,k-1)
  end
  return prob
end
(b)
   計算量のオーダーが O(3^n)と指数の形で表されるため
   n の増加に伴って計算量が膨れ上がるから
(c)
 (キ)1.0
 (ク)value(a,x-1,y,z)*w(x-1,y,z)
       +value(a,x,y-1,z)*l(x,y-1,z)+value(a,x,y,z-1)*d(x,y,z-1)
```

解説

(a)

i 勝 j 負 k 引を(i,j,k)と表そう。今(i,j,k)ならば一試合前は(i-1,j,k),(i,j-1,k),(i,j,k-1)のいずれかである。あとは一試合前の各状況に至る確率と今の一試合の結果の確率の積を足す。つまり確率を p(i,j,k)として漸化式は

```
p(i,j,k) = p(i-1,j,k)*w(i-1,j,k) + p(i,j-1,k)*l(i,j-1,k)
                                 +p(i,j,k-1)*d(i,j,k-1)
```

ひとつ前の状況の確率は自身の関数を呼び出し再帰的に求めている。解答例では面倒な条件分岐をしているが理由は(c)解説後半で。

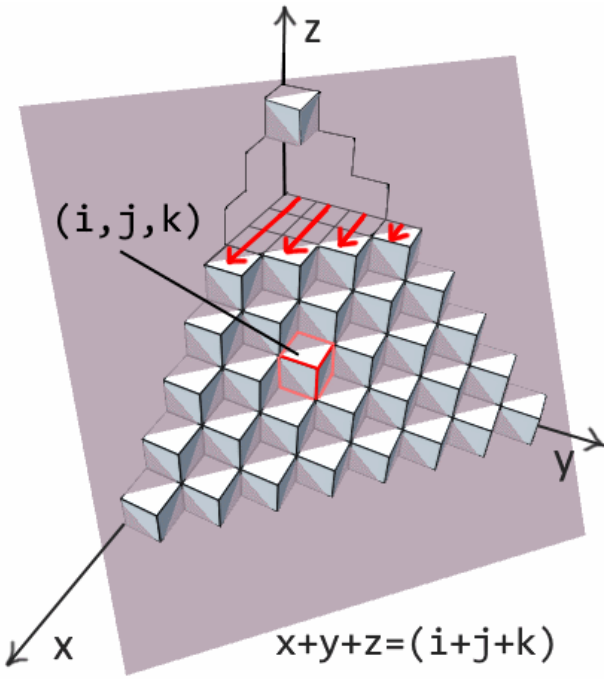
(b)

一つの関数から自身を三回呼んでいる。再帰的なアルゴリズムの計算量は指数の形で表されるように、実行時間の観点からは劣っている場合が多い。

(c)

初めに確率を格納する三次元配列を用意して `i+j+k=0` から順番に計画的に求めていく。関数 `make3d()`はその名の通り三次元配列を用意するもので、三次元でも多次元配列であるのは同じで二次元配列と扱いは同様。`value()`関数は三次元配列の値を取得するのだが、わざわざ渡されたインデックス `i,j,k` で条件分岐しているのはインデックスが負数だと `a[a.lngth+i]`と後ろから|*i*|番目を参照してしまうのを防ぐ目的である。代わりに `0` を返す。

三次元配列 `a` を用意したら順に詰めていくのだが、`a` のインデックス `x,y,z` は図のように小さい方から増えていく。関数 `value()`で `a` の適当な要素を取得し(a)と同じ漸化式を表現する。この時 `a` のインデックスに存在しない状況にあたる負数を渡してしまっても `0` が返ってくるので、掛けて `0` 足して`+0` で条件分岐せずとも問題なし。ただし、このままだと `x=y=z=0` つまり初めの状況の確率が `0` になってしまうので、この場合のみ特別に `1` を返す。最後に求める確率 `a[i][j][k]` を返しておしまい。



図は配列 `a` をブロックで空間的・視覚的に表現

＊補足

配列 `w[][][],l[][][],d[][][]`のインデックスに存在しない状況にあたる負数を渡していいのかは疑問である。適当な数値が返ればいいが、もし要素が空で参照に対し `nil` が返ると困るからである。ただ、(ク)を埋めるのに条件分岐で対応する余裕はないので適当に数値が返されると仮定した。この仮定の下では(a)はもっと楽に書ける。

```
def probability(i,j,k)
  if i<0 || j<0 || k<0
    return 0.0
  end
  if i+j+k == 0
    return 1.0
  end
  return  probability(i-1,j,k)*w(i-1,j,k)
    +probability(i,j-1,k)*l(i,j-1,k)+probability(i,j,k-1)*d(i,j,k-1)
end
```

また、前頁の図からも視覚的に分かるのだが、a[i][j][k]を求めるのに a[][][]の大きさは大き過ぎる。欲しい a[i][j][k]から逆に漸化式を辿れば、途中で必要な a[x][y][z]は 0≤x≤i,0≤y≤j,0≤z≤k だから a の大きさは(i+1)×(j+1)×(k+1)で十分。すると、コードは以下のように書き換えられる。

```
def fastProbability(i,j,k)
  a = make3d(i+1, j+1, k+1)
  for z in 0..k
    for y in 0..j
      for z in 0..i
        if x + y + z = 0
          a[x][y][z] = 1.0
        else
          a[x][y][z] = value(a,x-1,y,z)*w(x-1,y,z)
        +value(a,x,y-1,z)*l(x,y-1,z)+value(a,x,y,z-1)*d(x,y,z-1)
        end
      end
    end
  end
return a[i][j][k]
end
```

2012 年度　共通問題

問題1

解答例

(a)　5!=120 5 の階乗
(b)

```
def f(n)
  if n < 2
    return 1
  end
  s = 1
  for i in 2..n
    s = s * i
  end
  return s
end
```

(c)フィボナッチ数列の第 5 項 8
　　（ただし初項は n=0)
(d) (ア) 0
　　(イ) i <= n
　　(ウ) result[i-1] + result[i-2]
　　(エ) i + 1
　　(オ) n

解説

(a)
再帰で階乗を計算している。
(b)
再帰は反復処理でも表現できます。初期値 1 を与えてから 2..n を掛けていく。f(n)に倣って n<2 の場合は 1 を返すようにした。
(c)
関数内で引数を 1,2 小さくして再帰し、足している。よく見るフィボナッチ数列の項を再帰的に求めるアルゴリズム。初項の n の値にだけ注意しよう。
(d)
最初に各項を格納する配列を用意し、前から詰めていく。一度はやったことあるはず。

すいません、ほとんど解説していませんでした。

問題2

解答例

```
(a)
def minDeg (a)
  min = abs(a[0] - a[1])
  for i in 0..(a.length-2)
    for j in (i+1)..(a.length-1)
      deg = abs(a[i] - a[j])
      if deg < min
        min = deg
      end
    end
  end
  return min
end
```

```
(b)
def minDeg2 (a)
  min = a[1] - a[0]
  for i in 1..(a.length-2)
    if a[i+1] - a[i] < min
      min = a[i+1] - a[i]
    end
  end
  return min
end
```

解説

(a)

計算量はさておき、最も単純なアルゴリズムを考えてみよう。つまり、可能な全ての二数の組み合わせに対してその差を網羅的に調べればいい。すると、反復処理が組み合わせの数 n(n+1)/2 だけ存在すると考えられ、最高次数の項のみ注意して定数倍を無視し計算量のオーダーはちゃんと $O(n^2)$になりそう。変数 min に現時点での最小の差を格納し、より小さい差を見つけ次第代入する。ただし、初期値として a[0],a[1]の差を入れた。

(b)

配列が既にソートされているから、一般に i<j として a[i]と a[j]の差は j-i=1 の時が最小になる。よって、隣り合う要素の差だけ調べればいい。後は(a)と同様に実装する。反復処理は(n-1)回なので計算量のオーダーは O(n)になった。

解答例

(c)

（例 1）併合整列法

```
def mergeSort (a)
  deg = make1d(a.length)
  for i in 0..(deg.length-1)
    deg[i] = [a[i]]
  end
  n = deg.length
  while n > 1
    temp = make1d((n+1)/2)
    for i in 0..(n/2-1)
      temp[i] = deg[i*2] + deg[i*2+1]
    end
    if n%2 == 1
      temp[n/2] = deg[n-1]
    end
    deg = temp
    n = (n+1) / 2
  end
  return minDeg2(deg[0])
end
```

（例 2）CountingSort

```
def countSort (a)
  min = a[0]
  max = a[0]
  for i in 0..(a.length-1)
    if a[i] < min
      min = a[i]
    end
    if a[i] > max
      max = a[i]
    end
  end
  cnt = make1d(max-min+1)
  for i in 0..(a.length-1)
    index = a[i] - min
    cnt[index] = cnt[index] + 1
  end
  for i in 1..(cnt.length-1)
    cnt[i] = cnt[i] + cnt[i-1]
  end
  b = make1d(a.length)
  for i in 0..(a.length-1)
    index = a[i] - min
    cnt[index] = cnt[index] - 1
    b[cnt[index]] = a[i]
  end
  return minDeg2(b)
end
```

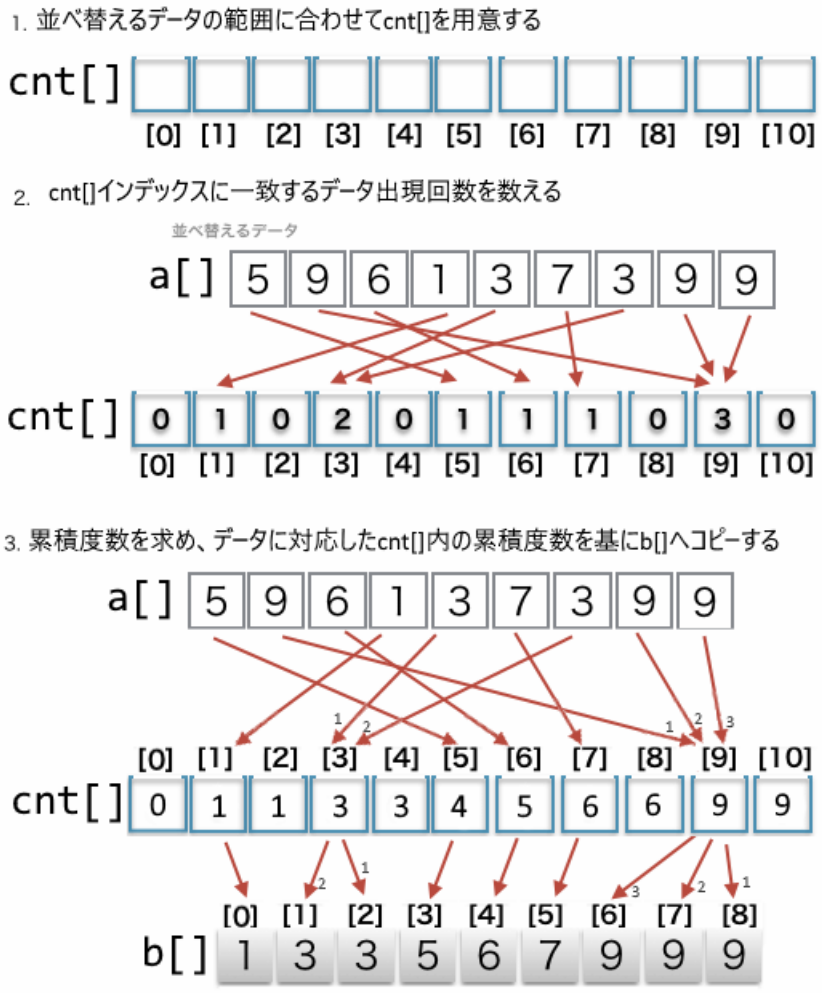
解説

計算量のオーダーに制限をかけ全てのコードを書かせるとはなかなか辛い。

ところで、(b)のようにソート済みの配列を用意すれば計算量O(n)のアルゴリズムで処理できることが分かっている。ならば、オーダーが(a)の $O(n^2)$ よりも優れた整列アルゴリズムを作ればいいじゃん、と思いましょう。整列法自体は多種多様に存在しますのでここでは二例だけ挙げてあります。一つ目は 2013 年度共通問題 1 にあった併合整列法です。詳しい説明はそちらを参照してもらうとして、こいつの計算量のオーダーはO(nlogn)ですから条件クリア。二つ目は数え上げ整列法でこちらは反復処理の回数がすべて n で収まり、オーダーが O(n)になる。ではこの中身を簡単に説明する。

発想としては、配列内である値が全体で何番目に小さいか分かれば、小さい順にソートした時その値の順番も分かる、ということ。具体的には、

まず、配列 a[]の要素を数え上げるためと配列 cnt[]と並べ替えた後の配列 b[]を用意。cnt[]を用いて a[]の値の出現頻度を数える。この時、a の値が cnt[]のインデックスになっている。次に c[]に格納したまま a の値の累積度数を求める。これで a の各値が全体で何番目か大体分かる。そして、累積度数に従って a[]の各値を b[]にコピーする。簡単のため、0≤(a[]の要素)≤10 と仮定すれば図のようにソートされてゆく。ところが、累積度数がコピー先のインデックスに直接は対応しておらず、またデータの重複も考慮する必要があるのだが、これは cnt[]内の累積度数をその都度-1 してから対応したインデックスの b[]へコピーすると上手くいく。



ただ、実際にはデータ a の範囲は不明で負数を含むかもしれないので、初めにデータの最小最大を求めて、データ a の最小値が cnt[]の先頭のインデックス 0 に対応するよう調整した。min=0 とすれば図と同じ状況になる。

問題3

解答例

(a) (ア) 1.0/akk
　　(イ) aik * a[k][j]
(b)
akk の値が 0 または 0 に近い値の場合、その逆数が極端に大きくなりのちの加算で誤差の原因になる
(c)
値 akk を取る時 k 列 k 行以降の成分 a[k][k]~a[row-1][k]中の絶対値が最大になる値を a[p][k]として、
k,p 行を入れ替えから akk に a[k][k]を代入する

```
def gj(a)
  .....
  for k in 0..(col-2)
    swap(a,k,maxrow(a,k))
    akk = a[k][k]
    ...........
end
```

(d)

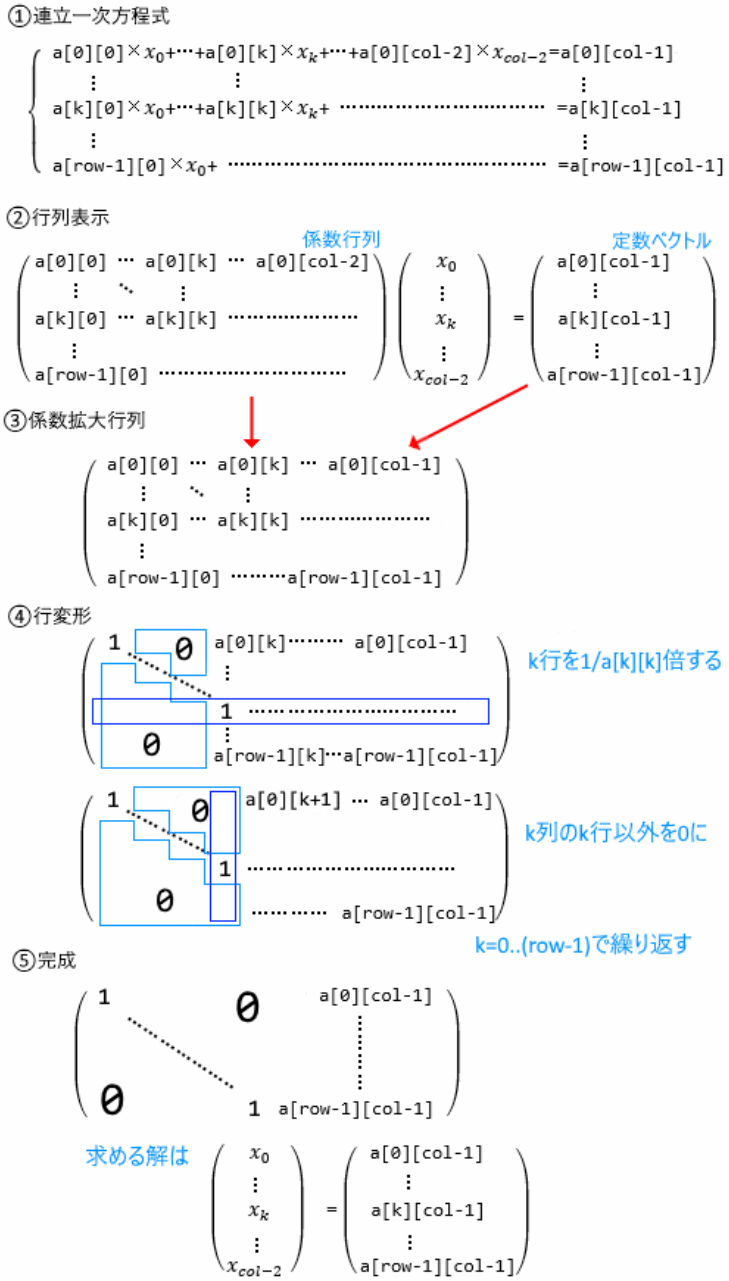
```
def gj(a)
  row = a.length
  col = a[0].length
  for k in 0..(col-2)
    swap(a,k,maxrow(a,k))
    akk = a[k][k]
    if akk == 0.0
      #不定方程式 or 不能方程式
      abort()
    end
    for i in 0..(col-1)
      a[k][i] = a[k][i] * 1.0/akk
    end
    for i in 0..(row-1)
      if i != k
        aik = a[i][k]
        for j in k..(col-1)
          a[i][j] = a[i][j] - a[k][j] * aik
        end
      end
    end
  end
  return a
end
```

解説

要するに係数拡大行列を簡約階段行列に変形して一次連立方程式を解くやつです。

(a)

この問題のように不能・不定方程式の場合を考慮しなければ、行列の左側部分が単位行列になります。方法として k 列に関して k 行成分を 1 に、それ以外を 0 になるように変形する作業を k=0,1..(row-1)まで続ければ、最後には対角成分 1 だけが残る。このとき各作業は、行を実数倍する(0 を除く)、行の実数倍を他行に足すという行基本変形に対応していると気づくでしょう。



そのためには図のように、まず k 行 k 列成分 a[k][k]で k 行の各成分を除算して対角成分を 1 にする。(ア)1.0/akk この時、商が返る整数型どうしの除算は回避するべきで、かつ数値 akk は整数型かもしれないのでもう一方は 1.0 と浮動小数型にしておきます。

次に i 行(i=0,1..(row-1),i≠k)に k 行の-a[i][k]/a[k][k]つまり(イ)-aik 倍を足すと対角成分以外は 0 になる。あとはこれを繰り返すと、最終的に行列の最右列を除く部分が単位行列に変形され最右列に解が現れる。

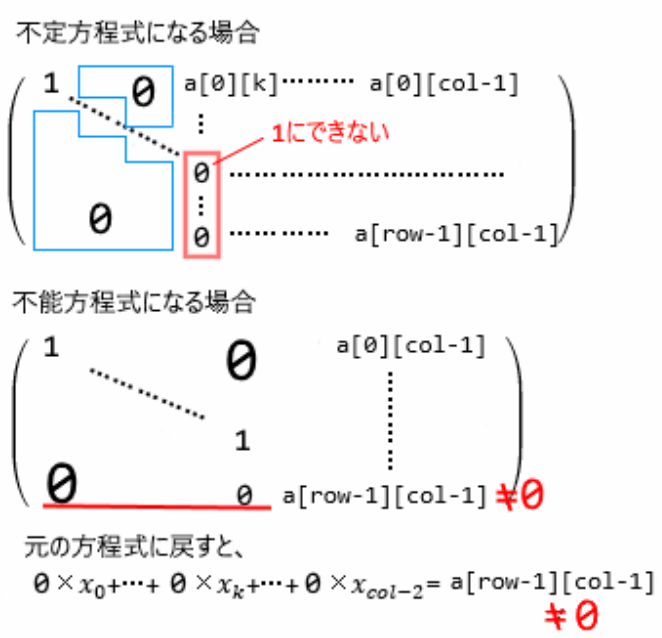
(b)

対角成分を 1 にするとき、その行の各成分を対角成分で除算する作業があるのだが、もし 0.0 または 0 に近い値で除算すると極端に大きい値になり得えます。
そして後に行の実数倍を他行に加算する段階で問題になる。
なぜなら極端に大小の異なる値を加減算すると誤差（桁落ち、情報落ち）を発生する可能性があるからです。

(c)
要するに除算する値 akk が大きければ問題ないということ。
そこで k 行 k 列成分を 1 にする前に k 列の k 行以降の成分を眺めてみて、絶対値最大の成分を p 行に発見したら k,p 行を入れ替えます。（これも行基本変形）そうすれば k 行 k 列成分が極端に小さいという事態は避けられそうです。以下の通り部分変更します。

```
....
  for k in 0..(col-2)
    swap(k,maxrow(a,k))
    akk = a[k][k]
....
```

(d)
一般に単位行列を目指して変形する過程で階段の下が 0 だけの列が出現すると解の自由度が増えて解が不定になります。これは(c)で改良した部分で探し出された最大値が 0.0 である場合に該当します。
次に係数行列の変形が終わった段階で最下行が 0 ベクトルでかつ右隣が 0 でないと不能方程式になります。0 との積はいくら足しても 0 で 0 以外にはなり得ませんからね。ただ、不能の場合もこの問題では不定と同じ方法で判断できます。（理由は以下の補足で）



＊補足
なぜなら、今回は不定と判断した時点で終了しかつ係数行列が正方であるため、最下行の変形まで不定判断が無いならば最後に係数行列の一番右下を 1 に変形しようと試みるはずです。もし、そこで不能判定されるならば少なくとも右下のやつは 0 です。
ただ、(b)で言及されている誤差を気にするなら浮動小数の等号比較は褒められないので、適当に最大許容誤差の大きさでも用意して　a[k][rowIndex] == 0.0 → Math.abs(a[k][rowIndex]) < (許容誤差)　と不等式で評価したほうが無難かもしれないが、この問題では等号でも正答だろう。

【変更部分】

```
....
  for k in 0..(col-2)
    rowIndex = maxrow(a,k)
    if a[k][rowIndex] == 0.0
      abort()
    end
    swap(k,rowIndex)
    akk = a[k][k]
....
```

問題4

解答例

(a) 1,0,0,2,2
(b) “11011”
(c) 文字列配列 a,b とその長さ m=3,5 以上の整数 n,空文字列 as=””,bs=””
を引数に渡して関数 post()を呼ぶと”0110000”が返る
(d) O(m^n)

解説

(a)

パズルです。頑張りましょう。発想としては、b[]の"0"だけでは無理そうなので"011"も使うと判断し、これに合わせるには a[]の"1"を二つ繋げると思う。以下省略

(b)

順に辿ればいいだけです。

このアルゴリズムの内容を翻訳すれば以下のようになるでしょう。

今、連結する文字列の配列 a[],b[]と現段階までで連結してできた文字列 as,bs があり、あと最大 n 回まで連結できる。この情報を引数に渡して呼び出し、as,bs を一致させようとあれやこれやする。最初の条件分岐でもし空文字でなく as==bs ならこれが欲しい文字列に他ならない。一致しないならば、あと何回連結できるか確認してもう無理なら目的の文字列は無かったと空文字を返す。まだ連結できるならば、k=0,1..(m-1)の各場合に対し文字列 a[k],b[k]を as,bs に連結してみる。上手くいくならばこれを再帰した結果の p に空文字以外が返ってくるはず、と考えて空文字でない文字列 p が得られるまで k を増やしていく。

(c)

小問(a)を解くという具体的な状況だけ考えればいいのです。

初めはまだ何も連結しておらず as="",bs=""、(a)の結果より５回連結するから n は５以上なら何でもいい。

そうすれば、関数 post(a,b,3,5,"","")は文字列"0110000"を返す。

(d)

一つの関数から自身の関数が m 回呼ばれるのが n 回繰り返されており、

結局 m 個の文字列を最大 n 個連結して出来る文字列をすべて調べるので全部でm^n通り調べたとも解釈できる。

よって計算量のオーダーは O(m^n)

2011 年度　共通問題

問題1

解答例

(a)　s(a,0,2) = 4, s(a,0,3) = -1, s(a,0,4) = 1
(b)　O(N)
(c)　O(N^2)
(d)

i	0	1	2	3	4	5	6	7	8	9
sum	8	4	0	2	6	1	6	9	2	10
t	8	8	8	8	8	8	8	9	9	10

(e)　O(N)

解説

(a)
　足しましょう
　注意したいのは s(a,i,j)の和に a[j]は含まれません
(b)
　N 回の反復がある
(c)
　全ての i,j の組み合わせ N(N+1)/2 通りを調べるので、計算量のオーダーは最高次の項のみ注目し定数倍も無視して O(N^2)
(d)
　ただコードを辿れば解けるは解ける。
　その中身は、直観的に解釈すれば途中までの和 sum が負になったら、その先の和には含まれないはず？詳細については後の補足で
(e)
　コードからも明らかに N 回の反復に収まっている。

＊補足
　(d)のアルゴリズムを読んでみた。が、問題文中の誘導がよく分かりません。
　どうしたら掲載コードのように実装されるのか不明です。
　そこで誘導は無視してコードを解釈してみる

(使う命題に証明を添えましたが、読むのが面倒なら事実として受け入れ読み飛ばしましょう)

まず以下の事実を確認しよう。
k=0..z(z>0)として s(a,0,k)が k=z で初めて負数になるならば、
a[z-1]は mss(a,0,upper)の和に含まれない。
（つまり mss(a,0,upper)=s(a,i,j)とすれば i<z≤j でない）

証明
(1)z=1 の場合
a[0]<0 なので明らかに a[0]は mss(a,0,upper)の和に含まれない。
(2)z>1 の場合
a[z-1]が mss(a,0,upper)の和に含まれる、つまり
mss(a,0,upper)=s(a,i,j)として i<z≤j と仮定する
定義より s(a,i,j) = s(a,i,z) + s(a,z,j)
また、仮定から s(a,i,j) ≥ s(a,z,j)
この二点より s(a,i,z) ≥ 0
すると s(a,0,z)<0 より s(a,0,i)<0
ところが i<z で s(a,0,z)が初めて現れる負数であることに矛盾

この事実を受け入れると何となくコードの意図が読めるはず。上の z が発見されると a[z-1]は mss(a,0,upper)には一切関わらず、その前後を同時に考える必要はありません。なので見つけたら a[z-1]で配列 a を分割して二つを別々に調べても良い。
そしたら後ろの方の部分の先頭から再び調べていき新たな z を見つけたら分割、と繰り返す。結果、そのような a の全要素で分割される個々の部分配列にのみ注目し mss()を別々に求めてその内の最大値を探せば十分と分かる。

そこで、配列 a の先頭から要素の和 sum を調べていき、sum<0 に転じたら次の要素から sum を足し直し始める。一連の過程での sum の最大値を t に格納する、という流れになっている。

ただし、分割された部分配列(と初めの配列)の mss()を知りたい時、先頭からの和のみ調べれば十分なのかは不明である。なので簡単に証明

先のような a の全要素で分割される任意の部分配列 a[lower..(upper-1)]について
mss(a,lower,upper) = s(a,i,j),lower ≠ i (lower<i≤j≤upper)と仮定する
(1)s(a,lower,i) = 0 の場合
s(a,i,j)= s(a,lower,i) +s(a,i,j) = s(a,lower,j)だから i=lower としても良い
(2)s(a,lower,i) > 0 の場合
s(a,lower,j) = s(a,lower,i) + s(a,i,j) > s(a,i,j)
mss(a,lower,upper) = s(a,i,j)に矛盾する
(3)s(a,lower,i) < 0 の場合
s(a,lower,k) < 0 になる k(lower<k<upper)は存在しないことに矛盾する
(まだ a を分割できるような要素が残っている)

よって先頭からの和を調べれば十分である。

問題2

解答例

(a) f(2,4) = 16, f(3,5) = 243
(b) c = a^b

解説

(a)
言われた通り計算しましょう
(b)
(a)で具体的にコードを辿る過程で気づくだろうが a の b 乗を求めている。
累乗ならばもっと簡単に for..end ループを用いて

```
def f(x,y)
  z = 1
  for i in 1..y
    z = z * x
  end
  return z
end
```

と書ける。一方で問題中では

・指数が偶数ならば指数から×2 だけ取り出して指数は半分、基数は二乗にする

$$X^{2Y}=(X^2)^Y$$

・指数が奇数ならば指数から+1 だけ取り出して指数は-1、基数を前に出す

$$X^{2Y+1}= X\cdot(X^{2Y})$$

を指数が 1 になるまで繰り返すことで実現している。
(ただし、ループ最後で必ず指数 y は-1 されるので終了条件は y=0)
なんで、こんに複雑なんだと面倒に思うかもしれない。しかし、これにはある利点がある。指数の減少の速さに注目すると、指数が奇数でない限り、積極的に半分にされていくので前の単純版で-1 し続けるよりも早く 1 になる。つまり計算量のオーダーの観点からは後者のほうが優れていて、具体的には O(y),O(logy)となる

問題3

解答例

(a)
乱数のように見えるが、実際には確定した関数で決定される値
(b)
周期が十分長い/値の分布が一様/値の出現に偏りがない
(c)

```
    -dy*x <= dx*(1-y) && -dx*(1+y) <= -dy*x
```

(d)
(イ)x = -x1　　(ウ)x = x1

解説

(a)(b)
そもそも乱数とは本質的に規則性も再現性もないはずなので、予測は不可能であり計算機で表現することもできない。(既出の情報から次が予測できたら乱数ではない！)
そこで周期が非常に長く、入力の変化に対して出力が無関係に見えるぐらい複雑に変化して、その値が一様に分布するような関数を用いて表現する。しかし、とは言っても確定した計算方法で求められるので予測は可能である。

(c)

移動前の点を(x_0, y_0)と表そう。穴の端点(0,1)(0,-1)と線分(x_0, y_0)-(x1,y1)の位置関係に注目して点(0,1)は線分の上側に、(0,-1)は下側に存在すれば穴から抜ける。(線分上を含む)　直線の方程式は

$dx(Y-y_0) = dy(X-x_0)$

0≤x, x1<0 より dx≠0 であり　　$Y = y_0 + \frac{dy}{dx}(X-x_0)$

これを用いて条件を数式で表現すれば　$-1 \leq y - \frac{dy}{dx} x_0 \leq 1$

ただし(ア)で不等式を評価する時、点 (x_0, y_0)の情報は x,y に残っているのを使う
また、大きさが 1 以下を二乗して値が 1 以下と表現すれば不等式の数を減らせる。

(d)

x1<0 でかつ(ア)の条件を満たさないならば、Y 軸の壁で反射するから-x1 が新しい x
x1 が負数でないならば、そのまま x1 が新しい x になる。

問題4

解答例

```
(a)maxGain(i,j) = gain(i,j)
        + max(maxGain(i-1,j-1),max(maxGain(i-1,j),maxGain(i-1,j+1)))
(b)  (3^m − 1)/2
(c)
(ア)0.0      (イ)gain(i,j)
(ウ)
   gain(i,j) + max(totalGain[i-1][j-1],
                        max(totalGain[i-1][j],totalGain[i-1][j+1]))
```

解説

(a)

今、位置(i,j)にいるなら、一つ前の位置(i-1,j-1)(i-1,j)(i-1,j+1)いずれかを経由したはずです。それぞれの場合の最大累積利益は

gain(i,j) + maxGain(i-1,k) (k=j-1,j,j+1)　　と表せます。

この中から最大値を選べばいいのです。

＊注意

場合によっては j-1=0、j+1=n+1 と存在しない位置になるが、その時 maxGain()は 0.0 になりかつ利益の値は正数なので最大値として選ばれる虞はない。

(b)

与えられた n,m の大小関係より 1 つの関数から 3 つ呼ばれるのが(m-1)回繰り返される。よって合計で

$1 + 3 + 9 + \ldots + 3^{m-1} = \frac{3^m - 1}{2}$ (初めの一回も忘れずに)

(c)

さっきまで関数 maxGain()で取得していた最大累積利益を配列 totalGain[][]で格納・取得します。初めに配列を用意したら、(b)と同様の漸化式に従って順に配列を埋めていく(イ・ウ)。ただし、あり得ない列番号 j=0,n+1 が渡されたら 0.0 を返すという maxGain()関数の仕様を再現するために、配列の大きさを両端+1 だけ多めに用意して、その両端には常に 0.0 が入っているように調整してある(ア)。

2010 年度　共通問題

問題1

解答例

(a) c[i][i] = 1

(b)

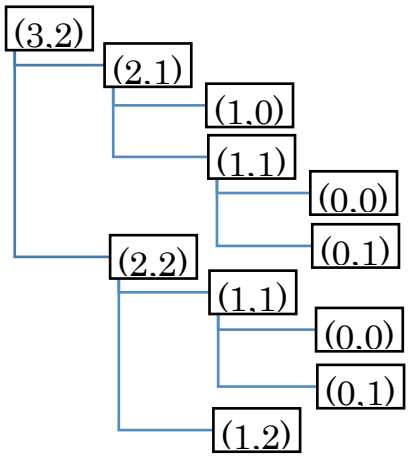
i	j	c[i][j]
2	1	2
3	1	3
3	2	3

(c)

代入の反復が合計で

1+2+..+(n+1)=(n+1)(n+2)/2 回あるから、計算量のオーダーは $O(n^2)$

(d)　簡便のため引数だけ書くと（上の方の枝から処理される）



(e)

前者は初めに配列を用意してから漸化式を用いて前から順に計画的に求めている
一方、後者は漸化式で目的のものから逆に辿る形で再帰的に求めている

解説

(a)

組み合わせの数 ${}_nC_k$ を求める。配列 c[i][j] が ${}_iC_j$ に対応しているのは明らかだろう。ここでは関係式 ${}_nC_k = {}_nC_k + {}_nC_k$ を用いる（${}_nC_n = {}_nC_0 = 1$ に注意）。ただし、この関係は 0≤k≤n の制約があり、k=0,k=n の場合にそのまま式に入れるわけにはいかないので ${}_nC_n = {}_nC_0 = 1$ は条件分岐で個別に設定する手間が要る。よって(イ) c[i][i] = 1。前の行で同様の操作 c[i][0] = 1 があることからも分かるだろう。

＊補足

${}_nC_k = {}_nC_k + {}_nC_k$ なぜ？と思ったら書き下してみれば

$$ {}_nC_k = \frac{n!}{(n-k)!k!} = \frac{n\cdot(n-1)!}{(n-k)!k!} = \frac{k(n-1)!+(k-k)(n-1)!}{(n-k)!k!} $$

$$ = \frac{(n-1)!}{(n-k)!(k-1)!} + \frac{(n-1)!}{(n-k-1)!k!} = {}_nC_k + {}_nC_k $$

(b)

順にコードを辿っていきましょう。

(c)

解答例の通りです。計算量のオーダーを求める時は、反復処理の回数に注目して n で表現し、その最高次の項のみ定数倍を無視して抜き出す。

(d)

書き出しましょう。こうして書き出してみると、再帰化的な関数は同じ計算を繰り返している部分があることに気づく。これが原因で再帰的なアルゴリズムが時間計算量

の観点からは劣っていることが多いのも分かる。

(e)

とりあえず、再帰関数なのか動的計画法なのかその辺りに言及すればいいでしょう。

問題2

解答例

```
(a)O(nm)
(b)
def intcount(a,b,c)
  x = -1
  j = -1
  for i in 0..(a.length-1)
    if x != a[i]
      x = a[i]
      j = j + 1
      b[j] = x
    end
    c[j] = c[j] + 1
  end
end
```

解説

(a)

コードを解釈すれば次のようになるでしょう。

変数 x が現在注目している配列 a の要素で、変数 j は x が配列 b に既に存在するか調べる時に参照する b のインデックスになる。この時 x に一致する要素を見つけるまで j を増やしながら b 中の正数を全て調べる。もし見つかれば while ループを脱した時 b[j]は何らかの正数で、見つからなければ 0 になっている。これで条件分岐し x が b に既出なら x=b[j]の出現数 c[j]を 1 増やし、新出なら新たに b に x を加える。

x が既出か否か調べる内側のループは平均すれば大体数値の種類だけ m 回、これが n 回反復するから合計 mn 回の反復がある

(b)

配列 a はソート済みなので同じ値は連続して出現します。つまり前から順に調べていきひとつ前と違う値が出現したらそれは必ず新出です。(a)において a のある要素が既出かどうか while ループで調べる手間が省けて、オーダーは O(n)で済む。

＊注意

a[0]は新出の値として処理したかったので x は a[0]と必ず異なる負数にした。また、この時 j が+1 されて j=0 になって欲しいから j=-1 と初期化してある。

問題3

解答例

(a)

0.1=0.00011001100…(2)のように二進数で表現すると無限小数になる値を
小さな桁の部分を丸めて近似的に表すことから生じる誤差

(b)

結果：false

理由：0.1 は二進数だと無限小数になるため丸め誤差が発生し、これを 3 倍すると誤差が強調されて 0.3 とは一致しないから

(c)

1.2345 - 1.2344 = 0.0001 のように
非常に近い値どうしを減算すると、有効桁数が減少することから生じる誤差

(d)

b^2に対して 4ac の値が極端に小さい場合、b と sqrt(d)の値の大きさが非常に
近くなる。すると、x1,x2 の一方は減算の過程で桁落ち誤差が発生する可能性が大きい。
そこでＢ君のアルゴリズムでは、もう一方の解から解と係数の関係を用いて求めることで、この誤差を回避しようとしている

解説

(a)(b)

他にも円周率π=3.1415…など無限小数になる。計算機のリソースは有限なので残念ながら無限の精度は表現できない。そこで限られた有限ビットで近似的に表されることから生じる誤差を丸め誤差と総称する。0.1*3 の例は教科書にもある有名な例。解答例程度の理解で十分だろうが、詳しくは補足で説明する。

浮動小数まとめ

計算機内で実数値を扱うときは浮動小数点表現を用いる。計算機で扱うからには0,1から成るビット列なのだが、符号部・指数部・仮数部の３つに分かれている。浮動小数点表現の中でも64bitのビット列を用いる表現を倍精度表現と言い、IEEE754規格によれば以下のように64ビットを使う。

s	c_1	c_2	...	c_{11}	m_1	m_2	...	m_{52}

s:符号　c_i(i=0,1..11):指数　m_k(k=1,2..52):仮数　　すべて0または1

これで表現される値は二進数表記で

$(-1)^s \times 1.m_1m_2..\ m_{52} \times 10^{c_1c_2..c_{11}-\mathrm{b}}$　(10$_{(10)}$でなく10=2$_{(10)}$に注意！)

ただし、bはバイアス値でb=1111111111$_{(2)}$=1023

小さな実数値を表すには、負の指数を表す必要がある。バイアス値とはそのための工夫であり、負の整数の補数表現を思い出せば分かりやすいかも。要するにバイアス値を0と考えるということ。

そこで0.1と0.3を実際に表現すると、(以下二進数表記、指数のみ十進表記)

$0.1_{(10)} = 1.\underline{1001100...1100}10011... \times 10^{-4_{(10)}}$

$0.3_{(10)} = 1.\underline{00110011...0011}0011... \times 10^{-2_{(10)}}$

52bitの仮数に収まるのは下線部で一つ下の桁を丸める（4捨5入ならぬ0捨1入)。

すると、以下のように計算機は表現する。

$0.1_{(10)} = 1.1001100...11010 \times 10^{-4_{(10)}}$

$0.3_{(10)} = 1.00110011...0011 \times 10^{-2_{(10)}}$

0.1*3はどうなるかというと、$3_{(10)}$=11なので

$$\begin{array}{rl} & 1.1001100...11010 \times 10^{-4_{(10)}} \\ \times) & \underline{\qquad\qquad 11 \times 10^{0_{(10)}}} \\ & 1.1001100...11010 \times 10^{-4_{(10)}} \\ & \underline{11.001100...11010 \times 10^{-4_{(10)}}} \\ & 100.110011...001110 \times 10^{-4_{(10)}} \end{array} = 1.\underline{00110011...001}10 \times 10^{-2_{(10)}}$$

仮数は52bitなので下の桁を丸めると、

$1.00110011...0100 \times 10^{-2_{(10)}}$

こうして0.1*3と0.3を比べると、仮数の最下桁が微妙に異なり、仮数の最下ビットの一つ分2^{-54}だけ差が生じるのが分かる。

(c)(d)

前問の(c)が(d)の誘導になっています。関数 fa()は解の公式をそのまま計算している。しかし、場合によっては計算過程で誤差を発生する可能性があります。

問題中の b=-100,$\sqrt{b^2-4ac}$ = $\sqrt{100^2-4}$ = 99.79998...のようにb^2に対して 4ac の値が極端に小さい場合、b と$\sqrt{b^2-4ac}$の値の大きさが近くなる。そして演算の結果が極端に 0 に近くなる加減算をすると有効桁数が落ちる桁落ち誤差が発生する。この場合は b<0 より -b+$\sqrt{b^2-4ac}$を計算する x2 に誤差が発生する。一方 x1 の方は問題ないので x2 は x1 と解と係数の関係から求めればこの誤差は回避できます。b>0 場合は x1 と x2 を逆にすればいい。

問題4

解答例

(a)

今、座標(m,n)にあるならば、一つ前の座標として(m,n-1)(m-1,n-1)(m-1,n-1)のいずれかを経由して来たはずであるから

$$T_{m,n} = T_{m,n-1} + T_{m-1,n} + T_{m-1,n-1}$$

ただし、m=0 または n=0 の場合はその座標までの経路は 1 通りしかないので

$$T_{0,0} = T_{0,i} = T_{j,0} = 1 \ (1 \le i \le M,\ 1 \le j \le N)$$

と定める

(b)

初めに、求めていく経路数を格納する二次元配列を用意してから
(a)の漸化式に従って座標が小さい順に経路数を数えては配列に格納していく
最後に求める値を返す。

解説

(a)

解答例の通りです。

(b)
コードを書けではなく、指針を示せとは中途半端に思える。
(a)の漸化式をそのまま使えば以下のように実装できます。

```
def rootCounter(m,n)
  cnt = make2d(m+1,n+1)
  for i in 0..m
    for j in 0..n
      if i == 0 || j == 0
        cnt[i][j] = 1
      else
        cnt[i][j] = cnt[i][j-1] + cnt[i-1][j] + cnt[i-1][j-1]
      end
    end
  end
  return cnt[m][n]
end
```

また、2011 年度共通問題４のようにあり得ない座標にアクセスしても 0 が返ってくれば問題ないので、次のように工夫すれば漸化式の場合分けを省ける。
座標(-1,N)~(-1,-1),(M,-1)~(-1,-1)の分だけ多く配列を用意して 0 で初期化しておく(初めの 1 通りを忘れずに(-1,-1)を 1 で初期化)。あとは漸化式に従って経路数をひたすら数えていくだけ。

```
def rootCounter(m,n)
  cnt = make2d(m+2,n+2)
  cnt[0][0] = 1
  for i in 1..m+1
    for j in 1..n+1
      cnt[i][j] = cnt[i][j-1] + cnt[i-1][j] + cnt[i-1][j-1]
    end
  end
  return cnt[m+1][n+1]
end
```

ただし、
関数 make2d()が返す二次元配列の各要素は 0 で初期化されている(のはず)と仮定