パターン認識入門

今回の話題：パターン認識

長大な列（例えば文章）から興味深い部分（例えばある文字列を含む部分）を取り出したい

- ある文字列を含むwebページを抽出
- プログラム中の特定の関数の呼び出しを
- DNAから面白そうな塩基配列を
  - 例えば特定の塩基をたくさん含む場所を
- スパムメールの識別
  - 「B-CAS」だけでなく「B-C@S」なども検出したい

簡単なパターン認識:文字列検索

- 文字列 s は、その一部として文字列 p そのものを含むか？　含むとしたら何文字目からか？

abc ∈ **ababcdba**

abc ∉ **abacdbac**

- 2ステップに分けて考える
  - 文字列 s の k 番目から始まる文字列は p か？
  - 上の条件を満たす「k 番目」は？

文字列検索のプログラム

```
def match(s,p)
  # sはpを含むか？
  m = Float::INFINITY
    # 含まれないときは∞を返す
  for i in 0 .. s.length()-1
    if submatch(s,p,i)
      # pはsのi番目からに一致するか確認
      m = i
    end
  end
  m
end
```

文字列検索のプログラム(続き)

```
def submatch(s,p,i)
  # pはsのi番目からに一致するか？
  f = true
  for j in 0 .. p.length()-1
    f = f && s[i+j] == p[j]
  end
  f
end
```

以下のプログラムでも良い(補足)

```
def submatch(s,p,i)
    s[i .. i + p.length() -1]  == p
    #sのi番目からp文字分がpと一致する？
end
```

文字列検索のプログラムの計算量

文字列 s の長さを m, p の長さを n とする

- **`Match`**はm回を**`submatch`**を繰り返す
- **`submatch`** は n 文字調べて結果を返す

➔ 時間計算量は**O(mn)**

この結果は満足か？

- nが小さい（実用上多い）場合は問題なさそう
- より高速なアルゴリズムもある
  - 例: Boyer-Moore 法

もっともっと複雑なパターン認識: アラインメント

文字列aと文字列bはどれくらい似ているか？

```
abcaa  bac              abcaa  bac
ab  acddba   （5点）      b  aacdbac （6点）
```

応用例:

- 遺伝子の類似度の計算
- コピペの抽出
- 編集された部分の抽出
- 文書修正時の最小の編集の計算

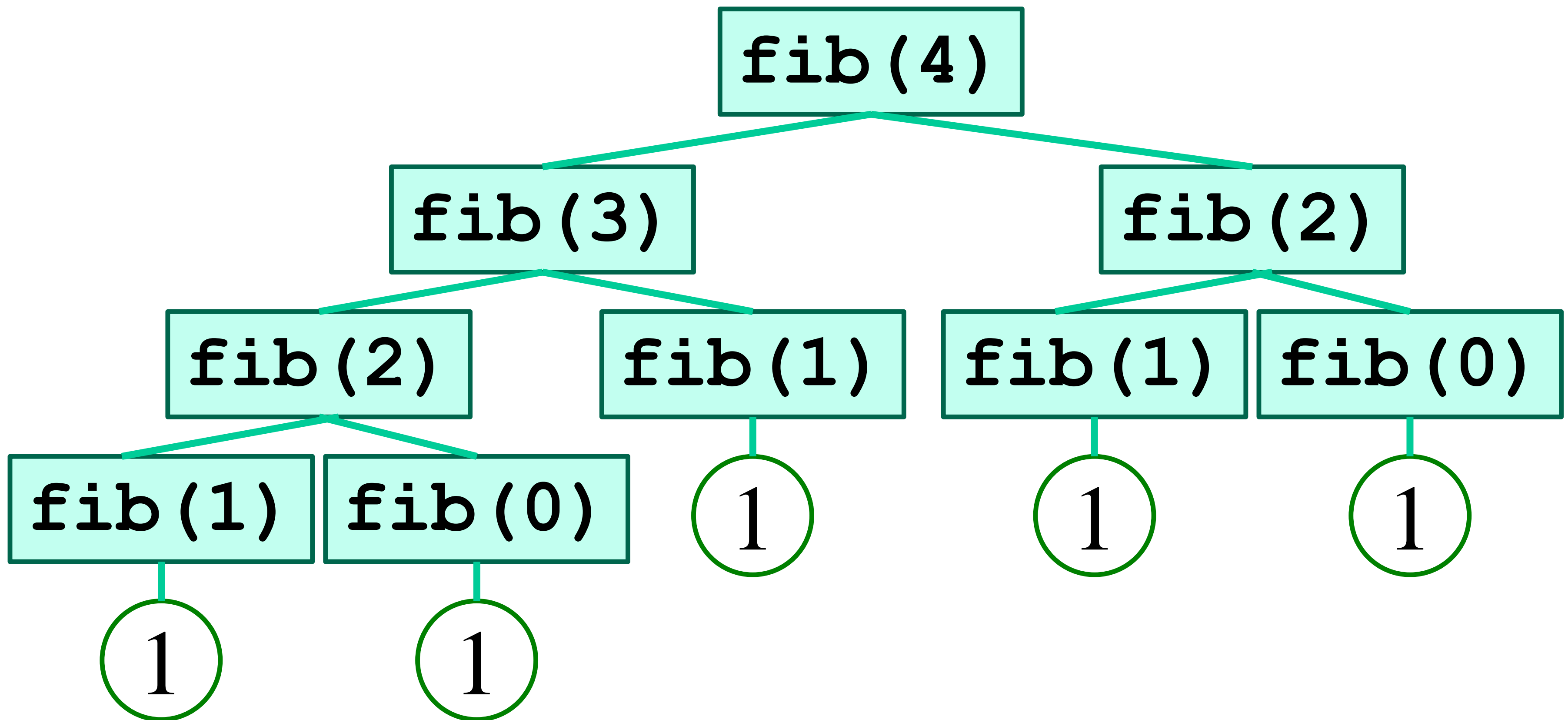
`fib`の漸化式（復習）

`fib`の漸化式:

- $fib(0) = 1$
- $fib(1) = 1$
- $fib(n) = fib(n-1) + fib(n-2)$

これを素直に実装した再帰関数は遅い……

fib(4)の計算の様子



fib(n)にはfib(n)回の再帰呼び出しが必要

動的計画法

素直な再帰関数を書いてしまうと同じ関数呼び出しを何度もやってしまって非効率
➔ 一回やった計算の結果は覚えておけば？

– fibの場合は「直前二つ」だけ覚えていれば良かったけど、一般的には？

動的計画法:
最終結果を求めるのに必要な値の「表」を作る

- 表が埋まった≒問題が解けた
- 表の既に埋めたマスの値は何度も使える

動的計画法の作り方

- 問題の答えを漸化式で表現する
  - 再帰関数を作る場合と同様
- 欲しい項を求めるのに必要な値を覚える表を用意する
- 表を漸化式の依存関係に従って順に埋めるプログラムを書く

動的計画法の適用例：`fib`

動的計画法によって`fib`を実装してみる

`fib`の漸化式：

- $fib(0) = 1$
- $fib(1) = 1$
- $fib(n) = fib(n-1) + fib(n-2)$

nが小さい方から順に計算

長さ$n+1$のテーブルを用意
i番目のマスは，$fib(i)$を保持

動的計画法によるfib

```
def fibt(n)
  # n番目のフィボナッチ数を求める
  table = make1d(n+1)
  for i in 1 .. n
    #nが小さい場合から順に
    table[i] = fill(table, i)
      #table[i] = fib(i) となるよう
      #tableを埋める
  end
  table[n]
end
```

動的計画法によるfib（続き）

```
def fill(table, i)
  #table[i] = fill(table, i) = fib(i)
  if i <= 1
    1
  else
    table[i-1] + table[i-2]
  end
end
```

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$

動的計画法の様子

`table`は以下のようなものとなる

0	1	2	3	4	5	6	7	8
1	1	2	3	5	8	13	21	34

これを左から右に順に埋めてゆく

➔ フィボナッチ数列をそのまま順に書く、
というアルゴリズムだと思えば良い

余談：他のアルゴリズムとの比較

	時間計算量	空間計算量
`fib`（再帰関数）	$O((\frac{1+\sqrt{5}}{2})^n)$	$O(n)$
`fibl`（ループ）	$O(n)$	$O(1)$
`fibm`（行列乗算）	$O(\log n)$	$O(1)$
`fibt`（動的計画法）	$O(n)$	$O(n)$

動的計画法は一般的なアプローチ。

必ずしもベストなやり方とは限らない

動的計画法の適用例2: 組み合わせ数

組合せ数の計算を動的計画法で実装する

nやkが小さい方から順に計算

- 漸化式:
  - ${}_{n}C_{k} = {}_{n-1}C_{k-1} + {}_{n-1}C_{k}$
  - ${}_{n}C_{0} = 1$
  - ${}_{n}C_{n} = 1$

$(n+1) \times (k+1)$のテーブルの各マス(i, j)を${}_{i}C_{j}$で埋める

組み合わせ数の動的計画法

```
def combination(n,k)
  # nCkを求める
  table = make2d(n+1,n+1)
  for i in 0 .. n
    for j in 0 .. i
      #nやkが小さい場合から順に
      table[i][j] = fill(table, i, j)
      #table[i][j] = iCj
    end
  end
  table[n][k]
end
```

組み合わせ数の動的計画法(続き)

```
def fill(table, i, j)
  # table[i][j] = iCj
  if j == 0 || i == j
    1
  else
    table[i-1][j-1] + table[i-1][j]
  end
end
```

$$_n\mathrm{C}_0 = 1$$
$$_n\mathrm{C}_n = 1$$
$$_n\mathrm{C}_k = {}_{n-1}\mathrm{C}_{k-1} + {}_{n-1}\mathrm{C}_k$$

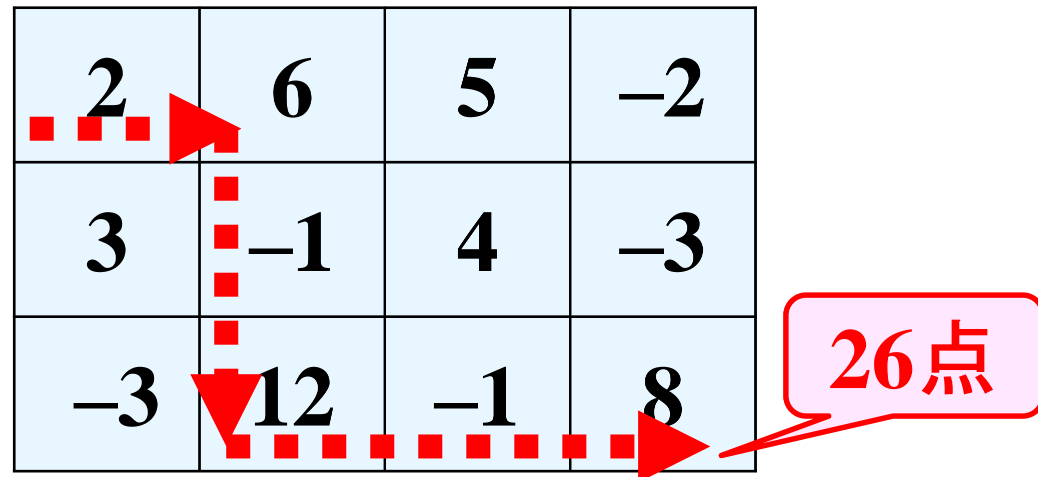
動的計画表による表

	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

➔ パスカルの三角形を上から下へ描いている

問題

$m \times n$の碁盤目状に部屋が繋がった迷宮があり、各部屋(i, j)に進入時に得られる得点$award[i][j]$が二次元配列$award$により決まっているとする。$award$を入力とし、$(0, 0)$から$(m-1, n-1)$に回り道せずに移動して得られる総得点の最大値を返す関数を示せ。



2	6	5	–2
3	–1	4	–3
–3	12	–1	8

問題のヒント

(i, j)の部屋までの得点の最大値$gain(i, j)$は以下の漸化式を満たす

- $gain(0, 0) = award[0][0]$
- $gain(0, j) = gain(0, j-1) + award[0][j]$
- $gain(i, 0) = gain(i-1, 0) + award[i][0]$
- $gain(i, j) = \max\{ gain(i, j-1) + award[i][j], gain(i-1, j) + award[i][j] \}$

DNAやタンパクの比較

- DNA --- 塩基（四種類）の配列
  - セントラル・ドグマ

    DNA → RNA → タンパク
  - 三塩基（コドン）が一つのアミノ酸に
  - 似ている配列は似ているタンパクに
  - 似ている配列は同じ祖先から進化
- タンパク --- アミノ酸（２０種類）の配列
  - 似ている配列は似た構造に
  - 似ている配列は似た機能を

アラインメント

- アラインメント

  二つ（複数）の文字列の比較
  - 音声認識・文字認識
  - DNAやタンパクの比較

GACGGATTAG と GATCGGAATAG

ギャップ　　不一致

```
GA-CGGATTAG
GATCGGAATAG
```

アラインメント

- ぴったり同じでなくとも、
  似ているかどうかを判定。
- スコア

  一致

  不一致

  ギャップ

  足し合わせる → アラインメントの類似度
- 類似度が最大の
  アラインメント（ギャップの入れ方）を求める。

最適化

例

スコア：　一致　+2
不一致　−1
ギャップ　−2

ATAG と **AAC** をアラインするには

ATA と **AA** のアラインメントに **G** と **C** を付加

ATA と **AAC** のアラインメントに **G** と **-** を付加

```
0   ATA    →   ATAG   −2
    AAC        AAC-
```

ATAG と **AA** のアラインメントに **-** と **G** を付加

```
0   ATAG   →   ATAG-   −2
    A-A-       A-A-C
```

アラインメントの動的計画法

- 二つの配列s0とs1の間の類似度
- a[i][j]： s0の部分配列s0[0],...,s0[i-1]と
  s1の部分配列s1[0],...,s1[j-1]の間の類似度
- a[i][j] = max{ a[i][j-1] + g,
  a[i-1][j-1] + q(s0[i-1], s1[j-1]),
  a[i-1][j] + g }

  g : ギャップのペナルティ（負の数）

  q(c, d) : c と d の類似度

  c と d が等しければ適当なスコア（正）

  似ていればそれなりのスコア

  似ていなければ不一致のペナルティ（負）

境界条件

- a[0][0] = 0
- a[0][j] = a[0][j-1] + g (j>0)
- a[i][0] = a[i-1][0] + g (i>0)

例えば g = -2

- 結局

  a[0][j] = j*g

  a[i][0] = i*g
- となる。要するに、ギャップばかり。

スコア：　一致　+2
不一致　−1
ギャップ　−2

ATAG
A−AC

		A	T	A	G
	0	−2	−4	−6	−8
A	−2				
A	−4				
C	−6				

スコア：　一致　+2
不一致　−1
ギャップ　−2

ATAG
A-AC

		A	T	A	G
	0	−2	−4	−6	−8
A	−2	2			
A	−4				
C	−6				

スコア：

一致	+2
不一致	−1
ギャップ	−2

ATAG
A-AC

		A	**T**	**A**	**G**
	0	−2	−4	−6	−8
A	−2	2	0		
A	−4	0			
C	−6				

スコア：一致 +2
不一致 −1
ギャップ −2

ATAG
A-AC

		A	**T**	**A**	**G**
	0	−2	−4	−6	−8
A	−2	2	0	−2	−4
A	−4	0	1	2	0
C	−6	−2	−1	0	

スコア：　一致　+2
不一致　−1
ギャップ　−2

ATAG
A−AC

		A	T	A	G
	0	−2	−4	−6	−8
A	−2	2	0	−2	−4
A	−4	0	1	2	0
C	−6	−2	−1	0	

ATA
A-A

ATAG
A-A-

ATA
AAC

スコア:		
	一致	+2
	不一致	−1
	ギャップ	−2

ATAG
A-AC

		A	T	A	G
	0	−2	−4	−6	−8
A	−2	2	0	−2	−4
A	−4	0	1	2	0
C	−6	−2	−1	0	1

ATA
A-A

ATAG
A-A-

ATA
AAC

ATAG
A-AC

align.rb

いわずもがな、
x と y の最大値

```
def max(x,y)
    if x>y
        x
    else
        y
    end
end
```

文字 c と文字 d の
類似度を返す

```
def q(c,d)
    if c == d
        2
    else
        -1
    end
end
```

一致

不一致

```
def align(s,t)
    m = s.length()
    n = t.length()
    a = make2d(m+1,n+1)
    a[0][0] = 0
    for j in 1..n
        a[0][j] = a[0][j-1] + g()
    end
    for i in 1..m
        a[i][0] = a[i-1][0] + g()
    end
```

境界条件

```
    for i in 1..m
        for j in 1..n
        # ここを埋めよ
        end
    end
    a
end
```

トレースバック

- 最大の類似度を与えるアラインメントを提示するために、配列の最後から、それまでの選択を振り返る（トレースバック）。
- gap0[i]は、s0のi番目の文字の前に入るギャップの数。0で初期化しておく。

スコア：　一致　+2
不一致　−1
ギャップ　−2

ATAG
A-AC

→i

↓j

		A	T	A	G
	0	←−2	←−4	←−6	←−8
A	↑−2	↖2	←0	↖←−2	←−4
A	↑−4	↖0 ↑	↖1	↖2	←0
C	↑−6	↑−2	↖↑−1	↖↑0	↖1

gap1
0
0
0
0

gap0	0	0	0	0	0

スコア： 一致 +2
不一致 −1
ギャップ −2

ATAG
A-AC

→i
↓j

		A	T	A	G
	0	−2	−4	−6	−8
A	−2	2	0	−2	−4
A	−4	0	1	2	0
C	−6	−2	−1	0	1

gap1
0
0
0
0

gap0	0	0	0	0	0

スコア： 一致 +2
不一致 −1
ギャップ −2

ATAG
A-AC

→i
↓j

		A	T	A	G
	0	−2	−4	−6	−8
A	−2	2	0	−2	−4
A	−4	0	1	2	0
C	−6	−2	−1	0	1

gap1
0
0
0
0

gap0	0	0	0	0	0

ATA
A-A

ATAG
A-AC

スコア：　一致　+2
不一致　−1
ギャップ　−2

ATAG
A-AC

→i
↓j

		A	T	A	G
	0	−2	−4	−6	−8
A	−2	2	0	−2	−4
A	−4	0	1	2	0
C	−6	−2	−1	0	1

gap1
0
0
0
0

gap0	0	0	0	0	0

スコア:		
	一致	+2
	不一致	−1
	ギャップ	−2

ATAG
A-AC

→i

↓j

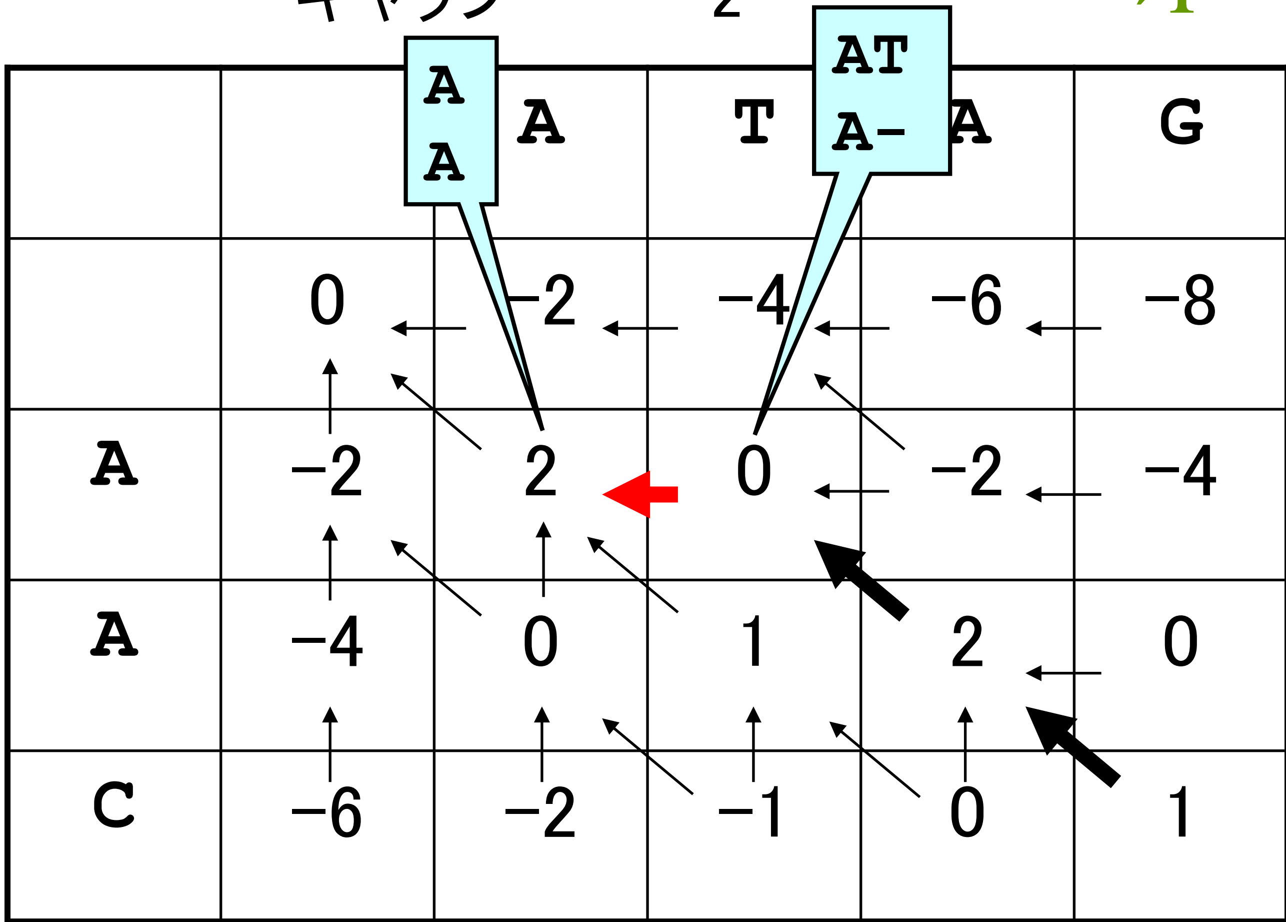
		A	T	A	G
	0	−2	−4	−6	−8
A	−2	2	0	−2	−4
A	−4	0	1	2	0
C	−6	−2	−1	0	1

gap1
0
1
0
0

gap0	0	0	0	0	0

スコア：　一致　+2
不一致　−1
ギャップ　−2

ATAG
A-AC



		A	T	A	G
	0	−2	−4	−6	−8
A	−2	2	0	−2	−4
A	−4	0	1	2	0
C	−6	−2	−1	0	1

gap1
0
1
0
0

gap0	0	0	0	0	0

スコア： 一致 +2
不一致 −1
ギャップ −2

ATAG
A-AC

→i

↓j

		A	T	A	G
	0	−2	−4	−6	−8
A	−2	2	0	−2	−4
A	−4	0	1	2	0
C	−6	−2	−1	0	1

gap1
0
1
0
0

gap0	0	0	0	0	0

```
def traceback(a,s,t)
  u = ""
  v = ""
  i = s.length()
  j = t.length()
  while i>0 || j>0
    if j>0 && a[i][j] == a[i][j-1] + g()
      u = "-"            + u
      v = t[j-1 .. j-1] + v
      j = j - 1 # go left
    else
      if i>0 && j>0 &&
          a[i][j] == a[i-1][j-1] + q(s[i-1], t[j-1])
        # 左上から求められた場合
```

```
      else
        if i>0 && a[i][j] == a[i-1][j] + g()
          # 上から求められた場合
        end
      end
    end
  end
  [u,v]
end
```

問題

- align.rb を完成させて、**`ATAG`** と **`AAC`** のアラインメントを求めよう。