プログラム設計とアルゴリズム
第12回 (12/13)

早稲田大学高等研究所　講師
福永津嵩

(前回の復習)問題のクラス

- 問題の難しさに応じて、問題をクラス分けする。ここでは、判定問題(YesかNoで答えられる問題)のクラスを考える。

- 最適化問題(最大値や最小値を求める問題)は判定問題ではないが、
最適化問題を判定問題に変更できることは多い。

- 「~~という場合の最小値を求めよ」
→
「~~という場合の最小値がWを超えるかどうかを判定せよ」

(前回の復習)クラスP

- クラスP:
  多項式時間アルゴリズムが存在する判定問題の集合

  例:
  1. 与えられた配列の中に特定の値が存在するかを判定する。
  2. 与えられた無向グラフが二部グラフであるかを判定する。

- 一方、次に紹介するハミルトンサイクル問題は、
  多項式時間アルゴリズムは見つかっていないが、多項式時間
  アルゴリズムが存在しないことも証明されていない。

  つまり、クラスPかどうかはまだわかっていない。

(前回の復習)ハミルトンサイクル問題

ハミルトンサイクル問題

図17.2右部

- なお、全ての「辺」をひとつずつ含む(同一の頂点を何度通っても良い)サイクルはオイラーサイクルと呼ばれる。

- オイラーサイクルの判定問題はクラスPである。

(前回の復習)クラスNP

- クラスNP:
  判定問題の答えがYesならば、そのYesである証拠を与えると、
  それがYesであることが多項式時間で検証できる判定問題の集合

- クラスPに属する問題は、明らかにクラスNPである。

- また、ハミルトンサイクル問題は、もし答えがYesであり、その答えであるハミルトンサイクルが与えられたならば、全ての頂点を通っていることは多項式時間で検証できるので、クラスNPである。

(前回の復習)P≠NP予想

- クラスNPとクラスPが同一の集合なのか、同一ではないのかはまだわかっていない。

- P = NPとはつまり、「Yesの証拠が与えられたときに多項式時間でYesであることを判定できる(クラスNPに属する)ならば、そもそも判定問題を多項式時間で解くことが出来る(クラスPに属する)」という主張。

- P ≠NPとはつまり、「クラスNPに属する問題でも、その判定問題を多項式時間では解けない問題が存在する。」という主張。

(前回の復習)NP完全・NP困難

- クラスNPに所属する問題の中で、最も難しい問題のクラスをNP完全と呼ぶ。ハミルトンサイクル問題は実はNP完全問題である
- 判定問題以外の問題について問題のクラスを考えたとき、NP完全問題に多項式時間帰着できる問題のクラスをNP困難と呼ぶ。

NP困難

- つまり、NP困難問題は、NP完全問題と同等かそれ以上に難しい問題となっている。

(前回の復習)メタヒューリスティクス

- NP困難な問題にたいして最適解を求める事は難しいことが多い。
一方実用上は、近似的に最適であれば十分であることも多い。
近似的に最適な解を発見する手法をヒューリスティクスという。

- メタヒューリスティクスとは、多様なNP困難問題に対して、
形式上どのような問題に対しても利用可能なヒューリスティクス
のことをいう。

- 焼きなまし法、タブー探索、遺伝的アルゴリズムなど
様々な手法が提案されている。

(前回の復習)山登り法(TSP)

1. まず最初にランダムに初期解xを生成し、そのスコアf(x)をbest scoreとする。

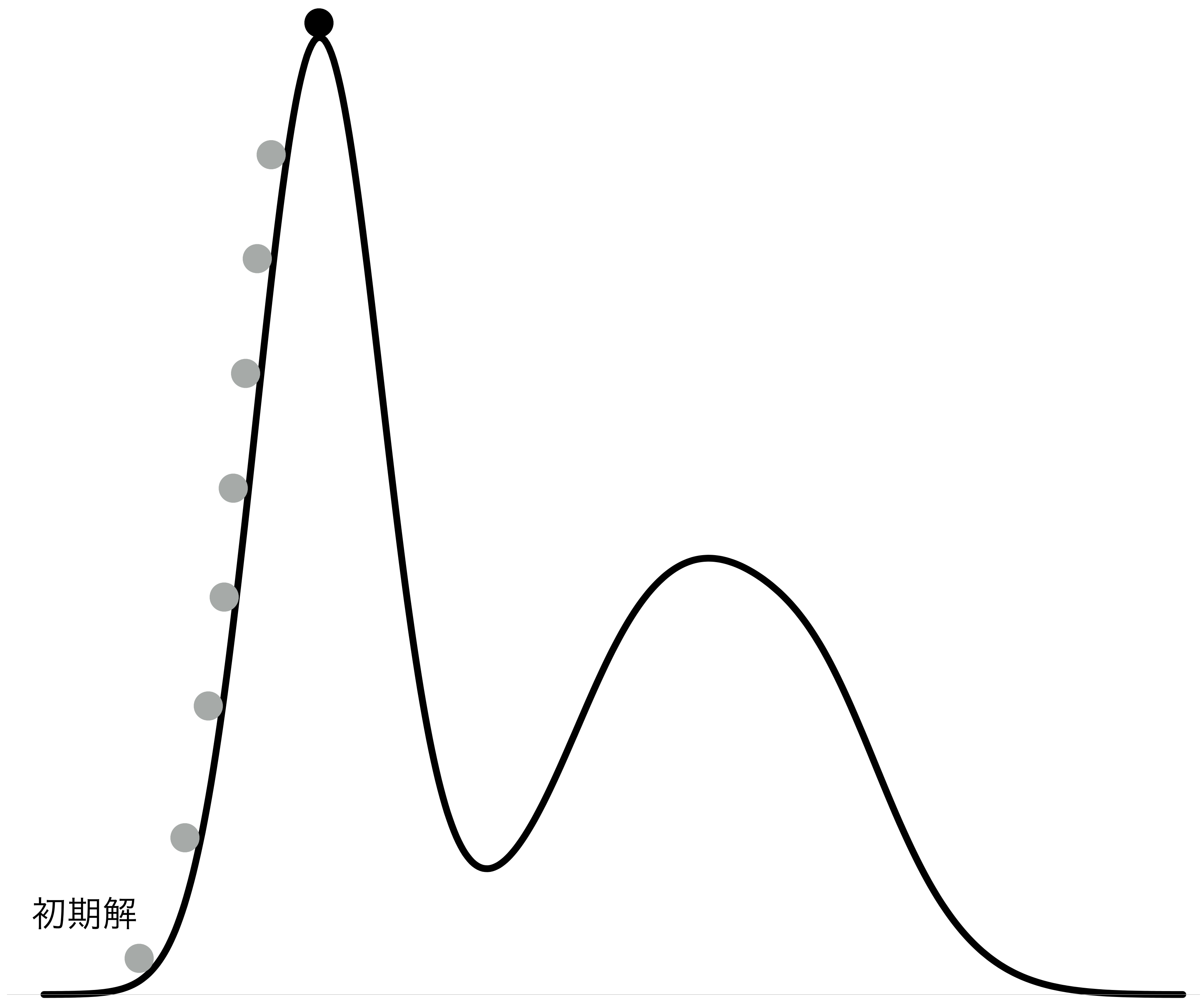
   (全ての都市を回るルートをランダムに生成し、その時間をスコアとする)

2. xの近傍x'の中で、最もスコアの良いx'に注目する。f(x')がbest scoreよりも良ければ、解とbest scoreを更新する。

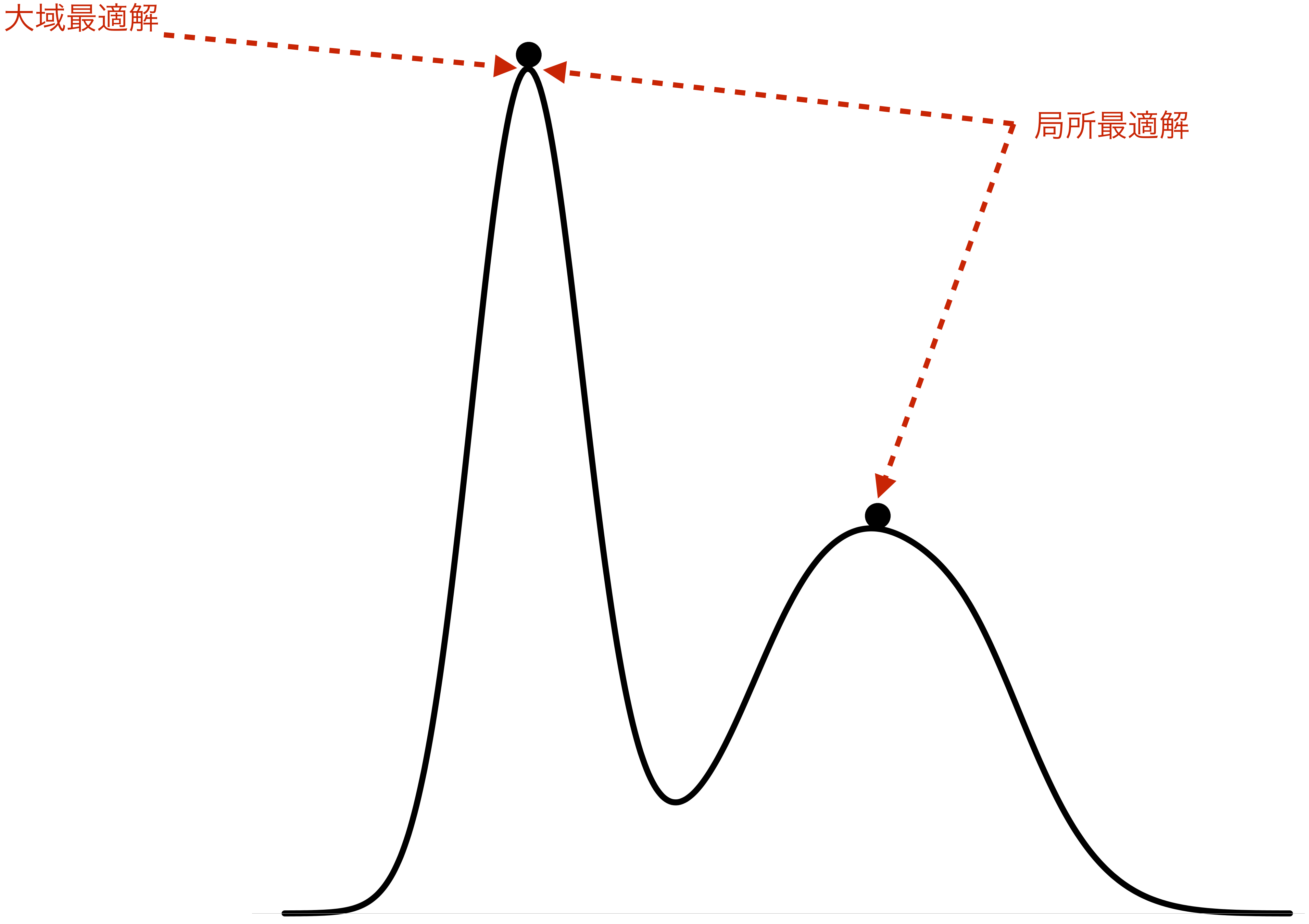
3. 2を繰り返す。

4. 解の更新が終了したら、探索を終了する。

(前回の復習)山登り法のイメージ図



(前回の復習)局所最適解と大域最適解



(前回の復習)焼きなまし法

- 現在の解よりも低いスコアの解があったとしても、一定確率以上で遷移する事で、より最適な解を発見する。

1. まず最初にランダムに初期解xを生成し、そのスコアf(x)をbest scoreとする。
温度Tと冷却率r $(0 < r < 1)$を設定する。
2. xの近傍の中からrandomにx'を選ぶ。
3. f(x')がbest scoreよりも良ければbest scoreをf(x')に更新する。
4. f(x')がf(x)より良ければ、x'をxに代入する。
f(x')がf(x)より悪い時は、$\exp((f(x') - f(x))/T)$の確率でx'をxに代入する。
5. 温度を更新する。$(T=rT)$
6. 2-5を繰り返す。
7. 一定回数繰り返すか、または温度が閾値を下回ったら繰り返しを終了し、
best scoreを出力する。

文字列解析

(復習)文字列間の距離を計算する

- 2つの文字列がどれくらい似ているのかを知りたいとする
(Webにおけるテキスト検索や、バイオインフォマティクスで必須)

- ハミング距離→長さが同じ文字列に対して、異なっている文字の数

HANABIとHAWAIIでは、

```
HANABI
||X|X|
HAWAII
```

のため、ハミング距離は2

(復習)文字列間の距離を計算する

- ではHANABIとHNABIAではどうか？

```
HANABI
|XXXXX
HNABIA
```

のため、ハミング距離は5となり、結構遠い。

- しかし、文字列だけ見ると非常に類似しているように見える。
この類似性は、ハミング距離では捉えきれないものなのかもしれない。

- ここで、新たな距離として、文字の削除と挿入を考えた
編集距離を考える。

(復習)文字列間の距離を計算する

編集距離

- HANABIとHNABIAでは、

```
HANABI-
|X||||X
H-NABIA
```

挿入

削除

のため、編集距離は2となる。

(復習)動的計画法:編集距離

編集距離を求める動的計画法:
dp[i][j]に、S[0:i-1]とTの[0:j-1]の編集距離を格納する

- 初期条件はdp[0][0] = 0
  そして、変更・削除・挿入の3つの場合を場合分けして考える。

変更操作 (S の i 文字目と T の j 文字目とを対応させる):

$S[i-1] = T[j-1]$ のとき: コストを増やさずに済みますので `chmin(dp[i][j], dp[i-1][j-1])` です.

$S[i-1] \neq T[j-1]$ のとき: 変更操作が必要ですので `chmin(dp[i][j], dp[i-1][j-1] + 1)` です.

(復習) 動的計画法:編集距離

削除操作 (S の i 文字目を削除)：

S の i 文字目を削除する操作を行いますので
`chmin(dp[i][j], dp[i-1][j] + 1)` です.

挿入操作 (T の j 文字目を削除)：

T の j 文字目を削除する操作を行いますので
`chmin(dp[i][j], dp[i][j-1] + 1)` です.

- この手続きの元、”logistic”と”algorithm”という2つの文字列を比較した場合の遷移は次のようなグラフで表現できる。
なお、計算量はO(|S||T|)

(復習)動的計画法:編集距離

図5.12

文字列マッチング問題

- ある長い文字列テキストTと、短い文字列パターンSが与えられた時に、Tの中に出現するパターンSの位置を全て出力せよ。

- 例)

```
T: ACTGCACGTCTGTACGTCAT
S: ACGT
```

答)

ACTGC<u>**ACGT**</u>CTGT<u>**ACGT**</u>CAT

0-originとすると、T[5]及びT[13]からSが始まっている。

brute-force法(力任せ法)

- Sを1文字ずつずらしながら、パターンとマッチするかを調べる。
  例)

```
T = ACATATAG
S = ATAT
```

- 1回目の比較:

```
ACATATAG
|X
ATAT
```

文字の総比較回数:0+2=2

brute-force法(力任せ法)

- Sを1文字ずつずらしながら、パターンとマッチするかを調べる。
  例)

```
T = ACATATAG
S = ATAT
```

- 2回目の比較:

```
ACATATAG
 X
 ATAT
```

文字の総比較回数:2+1=3

brute-force法(力任せ法)

- Sを1文字ずつずらしながら、パターンとマッチするかを調べる。
  例)

```
T = ACATATAG
S = ATAT
```

- 3回目の比較:

```
ACATATAG
  ||||
  ATAT
```

文字の総比較回数:3+4=7

出力:T[2]

brute-force法(力任せ法)

- Sを1文字ずつずらしながら、パターンとマッチするかを調べる。
  例)
```
T = ACATATAG
S = ATAT
```

- 4回目の比較:
```
ACATATAG
   X
   ATAT
```

文字の総比較回数:7+1=8

出力:T[2]

brute-force法(力任せ法)

- Sを1文字ずつずらしながら、パターンとマッチするかを調べる。
  例)

```
T = ACATATAG
S = ATAT
```

- 5回目の比較:

```
ACATATAG
    |||X
    ATAT
```

文字の総比較回数:8+4=12

出力:T[2]

brute-force法(力任せ法)の最悪計算量

- 下記のようなケースを考えると、その最悪計算量はO(|T| |S|)となることがわかる。
  例)

```
T = AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAT
S = AAT
```

- 1回の比較ごとに、|S|文字の比較が必要であり、それを|T|-|S|+1回分行わなければいけないため。

- とはいえこれは最悪ケースであり、平均的にはO(T)である。
  Tがランダム文字列であり、|A|を表れうる文字の数とすると、
  1回あたりの比較回数は、

  $1+(1/|A|)+(1/|A|)^2+\cdots+(1/|A|)^{|S-1|}$

KMP法

- Kunth-Morris-Pratt法
  力任せ法と同じく、Tの前から文字列のパターンマッチをしていく。
  例)

```
T = CTACTGATCTGATCGCTAGATGC
S = CTGATCTGC
```

- 1回目の比較:

```
CTACTGATCTGATCGCTAGATGC
||X
CTGATCTGC
```

- 2回目の比較を行う際に、力任せ法では1文字だけずらしたが、
  KMP法はずらす文字数を工夫する。

KMP法

- 1回目の比較:

```
CTACTGATCTGATCGCTAGATGC
||X
CTGATCTGC
```

- S[2]でミスマッチ→S[1](T)はマッチ
  つまりT[1]はCではないので、スキップして良い。
  T[2]はCかもしれないのでチェックする必要がある。

- つまり、1文字後を見るのではなく2文字後を見て良い。

KMP法

- 2回目の比較:

```
CTACTGATCTGATCGCTAGATGC
  X
  CTGATCTGC
```

- S[0]でミスマッチの場合はスキップできないので力任せ法と同じく1つ先に進む

KMP法

- 3回目の比較:

```
CTACTGATCTGATCGCTAGATGC
   ||||||||X
   CTGATCTGC
```

- S[8]でミスマッチ(おしい！)

- 何文字スキップ出来るか？Cが始まっている箇所ということで4文字はスキップ(5文字後を見る)できそうである。S[7..8]もTGであり先頭とマッチしているため、4文字スキップとなる。

KMP法

- 4回目の比較:

```
CTACTGATCTGATCGCTAGATGC
        ||||||X
        CTGATCTGC
```

- S[6]でミスマッチ。Cが始まっている箇所ということで
  4文字スキップ(5文字後を見る)できそうである。

- ただ、S[6]がTではないことがわかっているので、5文字目
  から開始してもマッチはしない。よって実は5文字スキップ出来る
  (6文字後を見る)。

KMP法

- 5回目の比較:

```
CTACTGATCTGATCGCTAGATGC
              X
              CTGATCTGC
```

- Tの中にSはありませんでした。

KMP法

- KMP法の概略:

  1. 与えられたSから表hを作成する。
     この時h[i]は、S[i]まで調べた時に何文字スキップして良いかを意味する。
     (表hはTには依存しないことに注意せよ)

  2. Tに対して、Sを前からパターンマッチしていく。
     パターンマッチに成功／失敗してSをずらす際には、1つずつずらすのではなく、i文字目まで調べたらh[i]だけずらす。

- `S:CTGATCTGC`
  の時の表h

h[0]	h[1]	h[2]	h[3]	h[4]	h[5]	h[6]	h[7]	h[8]
1	1	2	3	4	6	6	7	5

hの作成方法

- h[0]は明らかに1
- i > 0の時、h[i]はS[0..i-1]までマッチし、S[i]がマッチしなかった事を意味する。基本的にはiずらして良さそう。
- S[0..j] = S[i-j-1..i-1]であり、S[j+1] ≠ S[i] の時は特殊
  例) S[0..2] = S[5..7]であり、S[3] ≠ S[8]

```
CTACTGATCTGATCGCTAGATGC
   ||||||||X
   CTGATCTGC
```

この時はiずらすことはできない。

hの作成方法

- よって、
  $h[i] = i-1 - (\max j \text{ s.t. } S[0..j]=S[i-j-1..i-1], S[j+1] \neq S[i], j < i)$

- そのようなjがなければ、
  $S[0] \neq S[i]$ならば$h[i] = i$
  $S[0] = S[i]$ならば$h[i] = i+1$
  となる。

- このhは線形時間で構築出来る(KMPと似たような考えで行う)

KMP法の計算量

- 次の2つに着目する。
  I. 現在文字列Tの何文字目を見ているか
  II. 現在SはTの何文字目から比較を始めているか。

- IまたはIIがTの最終文字に達した時がアルゴリズム終了である。
  ここで、IもIIも増えはするが減りはしないことに着目する。

  Iについては、先の例で、3回目から4回目の比較を行う際に戻る必要がありそうに見えるが、4回目の最初のCTGはマッチしていることが保証されているので、4文字目のAから文字の比較を始めて良い
  (すなわちIが減ることはない。)

KMP法の計算量

- 文字の比較が成功した場合、Iが必ず1進む。
  文字の比較が失敗した場合は、IIが必ず1以上進む。

- よって1回の文字比較によりIまたはIIが必ず1進むので、
  2|T|回比較を行えば、IまたはIIのどちらかがTの最終文字に達する。
  すなわち、アルゴリズムが終了する。

- よってアルゴリズムの計算量はO(|T|)となり、
  hの構築時間も含めるとO(|S|+|T|)となる。

BM法

- KMP法は最悪計算量で見ると優れているが、実際には遅いことが多い
  そのため、Boyer-Moore法が利用されることがある。
- BM法では、パターンSを後ろからマッチさせる。

例)

```
T = CTACTGATCTGTTCGCTAGATGC
S = TAATAA
```

- 1回目の比較

```
CTACTGATCTGTTCGCTAGATGC
     X
TAATAA
```

BM法

- 1回目の比較

```
CTACTGATCTGTTCGCTAGATGC
     X
TAATAA
```

- 不一致が起きた際に、Gと不一致が起きていることに注目する。パターンSの中にGは存在しないので、SはGと被らないようにして良い。よって大幅にスキップできる。

BM法

- 2回目の比較

```
CTACTGATCTGTTCGCTAGATGC
           X
      TAATAA
```

- 不一致が起きた際に、Tと不一致が起きている。
  TはSの中で後ろから三文字目である。よって、そこがマッチするようにスキップする。

- 3回目の比較

```
CTACTGATCTGTTCGCTAGATGC
           |  X
        TAATAA
```

BM法

- このルールは不一致ルールと呼ばれる。
  すなわち、不一致が起きた時のテキストの文字がxである時、
  パターン中のxという文字のうち最後の文字の位置まで
  ずらすことができる。

- どれくらいずらすことが出来るかは、Sだけから計算できる。
  また、ずらす距離が後戻りしてしまう場合には、そのような後戻しは
  せず、1ずだけずらす。

- 例)

```
TATTAA      TATTAA
  X||    →       X|
TAATA         TAATA
```

BM法

- BM法ではさらに、接尾辞一致ルール(KMP法と類似)も用いる
  これは、後ろk文字が一致した場合、パタン中に同一のk文字が他にあれば、それにマッチさせるようにずらす方法である。

例)

```
ATATTAAGTAA
   X|||
TAAGTAA
```

↓

```
ATATTAAGTAA
    |||||||
    TAAGTAA
```

BM法

- 不一致ルールと接尾辞一致ルールのうち、大きい方をとって
ずらすことができる。

- KMP法は、テキストの文字を必ず一度は見なければいけないが、
BM法では、スキップによって場合によっては見なくて良い
という利点がある。

- 平均計算量は$O(|T|/\min(|S|,|\Sigma|))$となる。
ここで$|\Sigma|$はアルファベットサイズ
(スキップ量の期待値は$O(\min(|S|,|\Sigma|))$となるため。)

- ただし最悪計算量は$O(|T||S|)$(繰り返しが多い時など)
ただし、Turbo-BM法など$O(|T|)$にする改良法は存在する。

Rabin-Karp法

- ハッシュ関数を用いた文字列検索マッチング
- まず、文字列に対するハッシュ関数を定義する
  例)ローリングハッシュ
  文字列 $x = c_1c_2\cdots c_m$ としたとき、
  $hash(x) = (c_1a^{m-1}+c_2a^{m-2}+\cdots+c_ma^0)\ \%\ M$
- まずhash(S)を計算しておく。ここでSの長さはmとする。
  その後、hash(T[0..m-1])を計算し、値がhash(S)と同じなら同じ文字列かチェックする。違う値なら次に移動する。
- 次はhash(T[1..m])を計算する。これを繰り返し、T内の長さmの部分文字列全てに対してhash値の計算を行い、文字列の判定を行う。

Rabin-Karp法の例

- ローリングハッシュを用い、a=2, M=5とする。

```
S: 10101
T: 11001010110
```

- hash(S) = (16+4+1)%5 = 1となる。

- hash(T[0..4]) = hash(‘`11001`’) = (16+8+1)%5 == 0 よって次に進む

  hash(T[1..5]) = hash(‘`10010`’) = (16+2)%5 == 3

  hash(T[2..6]) = hash(‘`00101`’) = (4+1)% 5 == 0

  hash(T[3..7]) = hash(‘`01010`’) = (8+2)% 5 == 0

  hash(T[4..8]) = hash(‘`10101`’) = (16+4+1)%5 = 1

hash(S)と一致するので、SとT[4..8]が一致するかを調査する。

Rabin-Karp法の計算量

- ローリングハッシュの計算では、愚直に計算を行うと
  1回あたりO(|S|)かかる。そのため、全計算量がO(|S||T|)になる。
  ただし、次のような計算の工夫が出来る

- hash(T[0..m]) = $(t_0 a^{m-1}+t_1 a^{m-2}+\cdots+t_{m-1} a^0)$ % M
  hash(T[1..m+1]) = $(t_1 a^{m-1}+t_2 a^{m-2}+\cdots+t_m a^0)$ % M
  = ((hash(T[0..m]) - $t_0 a^{m-1}$) * a + $t_{m+1} a^0$) % M

- これにより、最初のhash(T[0..m])以外の文字列は定数回の計算ですむ。

- ただしハッシュ値の衝突が毎回起こったとすると、一致チェックが
  毎回起こることになり、この計算量はO(|S|)なので、結局
  最悪計算量はO(|S||T|)となる。

まとめ

- 文字列の編集距離は動的計画法によって計算可能であり、
  その計算量はO(|S||T|)となる。

- 文字列マッチング問題のアルゴリズムとして、
  力任せ法、KMP法、BM法、Rabin-Karp法の4つの手法を紹介した。
  最悪計算量としてはKMP法がO(|S|+|T|)と良いが、
  実用上はBM法などの方が高速になることが多い。