3.アルゴリズムと計算量

アルゴリズムとプログラム

- プログラム
  - 特定のプログラミング言語での計算手順の記述
- アルゴリズム
  - プログラミング言語に依存しない計算手順
  - 例：連立一次方程式や、最大公約数の解法
  - 正当性、停止性、計算量などについて考える

復習：最大公約数を求める

- ユークリッドの互除法
  - a, b (a ≧ b) について、a の b による剰余を r とすると、a と b との最大公約数は b と r との最大公約数に等しい
  - gcd(300, 24) = gcd(24, 12) = 12
  - greatest common divisor

ユークリッドの互除法

- 入力:2つの正整数$a, b (a \geq b > 0)$
- 出力:a, b の最大公約数
- 1. aをr_0とし、bをr_1とする。
- 2. r_0をr_1で割る。その商をq_1、余りをr_2とする。
- 3. 次に、r_1をr_2で割る。その商をq_2、余りをr_3とする。
- 4. このようにして割り切れるまで、繰り返す。

  r_nが最大公約数。

$$r_0 \div r_1 = q_1 \ldots r_2$$

$$r_1 \div r_2 = q_2 \ldots r_3$$

$$r_{n-2} \div r_{n-1} = q_{n-1} \ldots r_n$$

$$r_{n-1} \div r_n = q_n$$

正当性

- ちゃんと正しい結果を出すか

  $r_0 \div r_1 = q_1 \ldots r_2$ より $r_0 = r_1 \times q_1 + r_2$
- (r_0とr_1の最大公約数)=(r_1とr_2の最大公約数)
- (r_1とr_2の最大公約数)=(r_2とr_3の最大公約数)
- …
- (r_{n-2}とr_{n-1}の最大公約数)=(r_{n-1}とr_nの最大公約数)=r_n
- よって(aとbの最大公約数)= (r_0とr_1の最大公約数)=r_n

停止性

- ちゃんと計算が停止するか

  $r_0 \div r_1 = q_1 \ldots r_2$ より $r_1 > r_2$　同様に $r_1 > r_2 > r_3 > \ldots > r_n > 0$

- 非負整数の減少列は無限に続くことはないので必ず停止する

計算量

- 計算にかかる手間
- 複数のアルゴリズムがあるとき、その優劣を比較したい
  - 計算時間はコンピュータの性能によって異なる
  - 入力値の大きさに対する、基本的な命令の個数や演算の回数で比較するのが一般的

例:定積分の計算(p.70)

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    int k,n;
    double t,x;
    scanf("%d", &n);
    t = 0.0;
    for(k=1;k<n;k++){
        x = ((double)k)/n;
        t = t + sqrt(1.0-x*x);
    }
    t = t/n;
    t = t+1.0/(n*2);
    printf("%lg¥n",t);
}
```

入力値nが大きいほどfor文の繰り返し回数大
掛け算　n回
割り算　n+1回
平方根　n-1回

$$T_n = \frac{1}{2n} + \frac{1}{n}\sum_{k=1}^{n-1}\sqrt{1-\left(\frac{k}{n}\right)^2}$$

計算のオーダー

- 入力のサイズが大きくなるに従って、計算の手間がどのように増大していくか
- $\lim_{n\to\infty}\frac{g(n)}{f(n)}=c$ となるcが存在するとき、関数gはO(f)に属する　$g \in O(f)$
- 一番内側の繰り返し計算の回数を数える
- 先の定積分の例はO(n)
- 他に$O(n^2)$や$O(n \cdot \log n)$のアルゴリズムがある

空間計算量・時間計算量

- 空間計算量
  - どれくらいメモリを使うか
- 時間計算量
  - どれくらい計算の手間がかかるか

- 空間計算量が時間計算量を超えることはない

アルゴリズムのオーダーの比較

- 最大公約数を求めるアルゴリズム
  - 1から順に割り算をして最大の約数を探す(素朴)
  - ユークリッドの互除法(複雑)
- 計算結果としては同じになるはず
- だが、素朴なアルゴリズムは時としてすごい無駄な計算をしている(時間計算量のオーダーが大きく異なる)場合がある

1から順に割るアルゴリズム

```
#include <stdio.h>
int main(void){
    int x,y,N,i,ans;
    scanf("%d", &x);
    scanf("%d", &y);
    if (x > y) N = y;
    else N = x;
    for (k = 1; k <= N; k++)
        if ((x % k == 0) && ((y % k) == 0)){
            ans = k;
        }
    printf("%d¥n", ans);
}
```

xを入力

yを入力

xとyの小さい方をNに入力

1からNまで繰り返す

xもyも割り切れたらそれをansに

計算結果は正しいが、xとyが大きな値だと、
for文の繰り返し回数が大きくなってしまう

入力される整数の小さい方をNとすると、
繰り返し回数はN
オーダーはO(N)

計算時間

- 入力が10倍になると計算時間も10倍

x	y	実行時間(秒)	2進数桁数
10,000,000	10,203,040	0.450	24
100,000,000	102,030,400	3.074	27
1,000,000,000	1,020,304,050	29.402	30

6.83倍 +3

9.56倍 +3

6.83倍≒2^3

9.56倍≒2^3

- 10億で約30秒、1兆で約20時間,1京で約57年
- 入力Nの2進数桁数lの指数関数のオーダ $O(2^l)$

ユークリッドの互除法

```
#include <stdio.h>
int main(void) {
    int x,y,r;
    scanf(“%d”, &x);      xを入力
    scanf(“%d”,&y);       yを入力
    r = x % y;            割り算
    while (r > 0) {
        x = y;            前回の割る数をxに
        y = r;            前回の余りをyに
        r = x % y;        割り算
    }
    printf(“%d¥n”,y);
}
```

xとyが大きな値でも、割り算でどんどん値が
小さくなるから、素朴な方法より早く計算できそう
でも案外手間取る場合もある(144と89など)

ユークリッドの互除法で手間取る場合

- 入力された数の最大公約数が1 ←計算が最後まで終わらない
- 割り算の商が1 ←余りの値がなかなか小さくならない

$$144 \div 89 = 1 \ldots 55$$
$$89 \div 55 = 1 \ldots 34$$
$$55 \div 34 = 1 \ldots 21$$
$$34 \div 21 = 1 \ldots 13$$
$$21 \div 13 = 1 \ldots 8$$

$$13 \div 8 = 1 \ldots 5$$
$$8 \div 5 = 1 \ldots 3$$
$$5 \div 3 = 1 \ldots 2$$
$$3 \div 2 = 1 \ldots 1$$
$$2 \div 1 = 2$$

他にも

$$a_2 \div a_1 = 2$$
$$a_3 \div a_2 = 1 \ldots a_1$$
$$a_4 \div a_3 = 1 \ldots a_2$$
$$a_{n+1} \div a_n = 1 \ldots a_{n-1}$$

を満たす2数は手間取る

- これはフィボナッチ数列

$$\begin{cases} a_0 = a_1 = 1 \\ a_{n+1} = a_n + a_{n-1} \end{cases}$$

最悪の場合を考える

- ユークリッドの互除法で最悪の場合はフィボナッチ数列の2数を与えたとき
- フィボナッチ数列の一般項 $a_n = \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right\}$

$$\sqrt{2}^n \le a_{n+1}$$

$$\log_2 \sqrt{2}^n \le \log_2 a_{n+1}$$

$$n \le 2\log_2 a_{n+1}$$

n(繰り返し回数)は$2\log_2 N$以下

- 最悪の場合の時間計算量はO(logN)

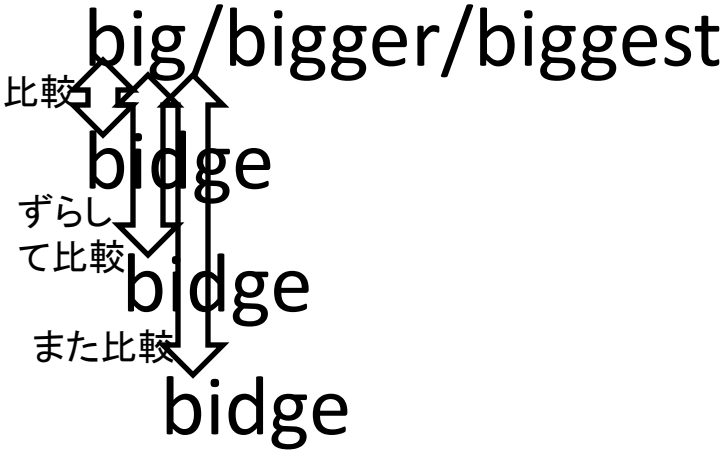
文字列の照合

- テキストの中に特定の文字列(パターン)が含まれているか探すこと
- 例:big/bigger/biggestの中にパターンbiggが出てくるか探したい
- big/bigger/biggest
- bigg
- bigg
- 1文字ずつずらして比較

比較

比較

照合回数を減らしたい



テキストの長さをlt,
パターンの長さをlpとすると
ずらす回数はlt-lp回

改良版(ボイヤー・ムーアのアルゴリズム)

・パターンを左→右にずらし、照合は右→左へ

big/bigger/biggest

一致しない

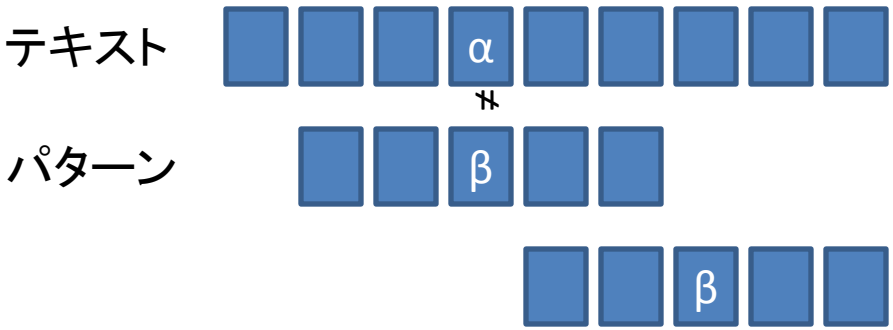
bidge

/はパターンに含まれていない
→/の次までずらしていい

bidge

どれくらいずらしていいか

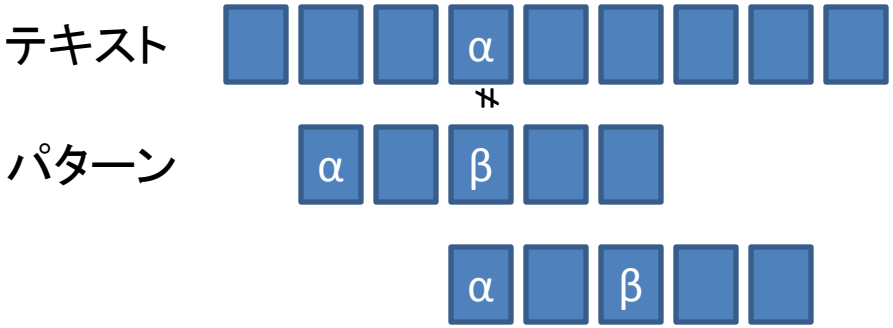
- パターンに含まれていない文字→その文字の次までずらす



αがパターン中に含まれていないならひとつずらしても駄目なのは明らか

ここまでずらしてからまた照合

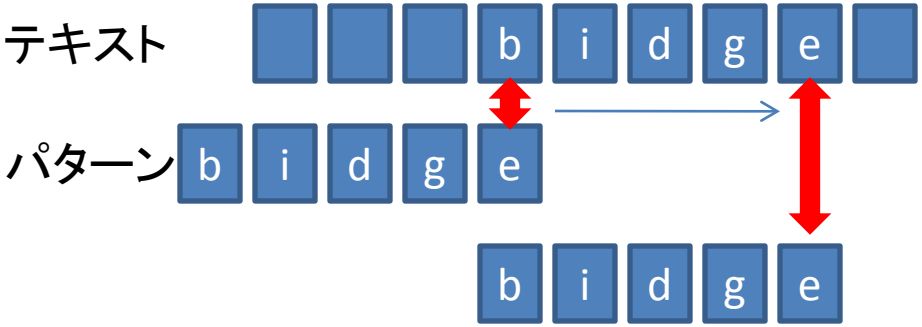
- パターンに含まれている文字→その文字とパターン中の文字が揃うようにずらす



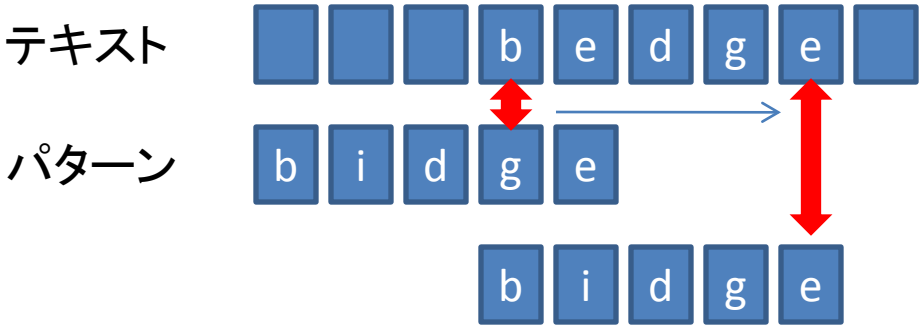
ここまでずらしてからまた照合

比較位置の移動量

- 「パターンの移動量」でなく、「比較位置の移動量」を表にする



bで不一致の場合
↕を4つ右に移動し
そこにパターンの右端がくるようずらす
(この場合パターンは4つずれる)



Bで不一致の場合
↕を4つ右に移動し
そこにパターンの右端がくるようずらす
(この場合パターンは3つずれる)

比較位置の移動量の表

- パターンがbidgeの場合

b	i	d	g	e	他の文字
4	3	2	1	5	5

bで不一致なら↕を4つ移動

- パターンに依存、テキストには依存しない

表をどう作るか

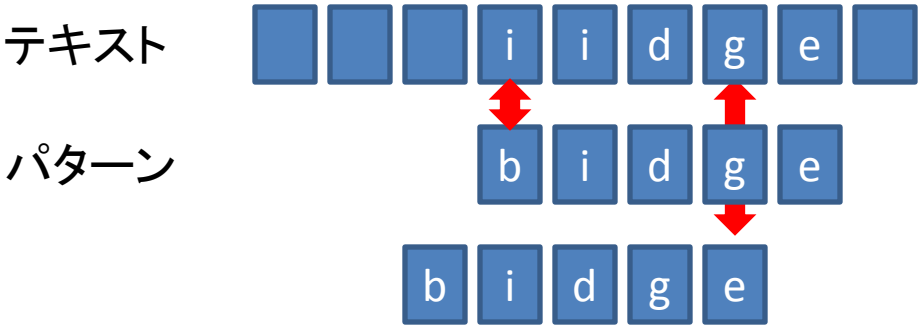
b	i	d	g	e	他の文字
4	3	2	1	5	5

- パターンの長さをnとする
- 他の文字、パターン末尾の文字 : n
- それ以外の文字:左からn-1,n-2,…,1

移動がうまくいかない場合

- 表の通りにすると逆行してしまう

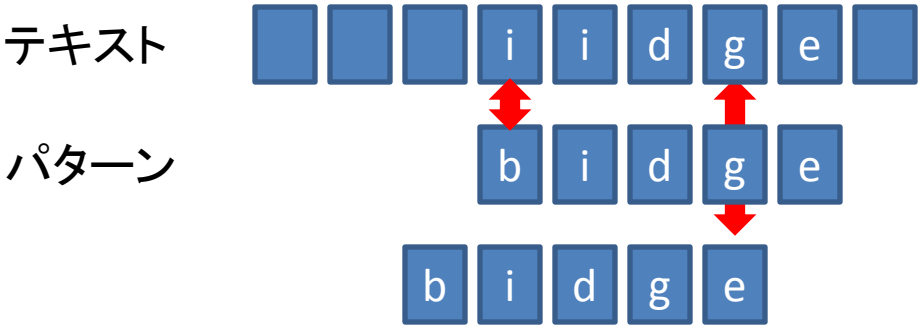
→右にひとつずらすことにする



表ではiの欄は3なので
逆にずれてしまう

ホールスプールのアルゴリズム

- 不一致文字に注目するのではなく、パターン一番右の文字と比較したテキスト文字に注目



表ではiの欄は3なので
逆にずれてしまう

- ボイヤ・ムーアのような逆行がない

ボイヤー・ムーアのアルゴリズム(簡略版)のプログラム例

```
#include <stdio.h>
#include <stdlib.h>
#define PATTERN_MAX_SIZE 1024
#define TEXT_MAX_SIZE 500000
#define SKIP_MAX_SIZE 128
int main(int argc, char *argv[]){
  char pattern[PATTERN_MAX_SIZE];
  char text[TEXT_MAX_SIZE];
  int psize, tsize;
  int i, j;
  int skip[SKIP_MAX_SIZE];
  int found;
  int ch;
```

パターン最大
テキスト最大
対応表最大
パターン配列
テキスト配列
パターン長、テキスト長
対応表配列

pattern:パターン、psize:パターン長①
text:テキスト、tsize:テキスト長②

```
for (ch = 0; ch < SKIP_MAX_SIZE; ch++)
  skip[ch] = psize;
for (j = 0; j < psize-1; j++)
  skip[pattern[j]]=psize-(j+1);
found=0; i=psize-1;
while (i<tsize && !found){
  found=1;
  for(j=0; j<psize; j++){
    if(text[i-j]==pattern[psize-1-j])
     continue;
    else {
    found=0;
    if(skip[text[i-j]]-j > 1)
      i += skip[text[i-j]]-j;
    else i += 1;
    break; }}}
```

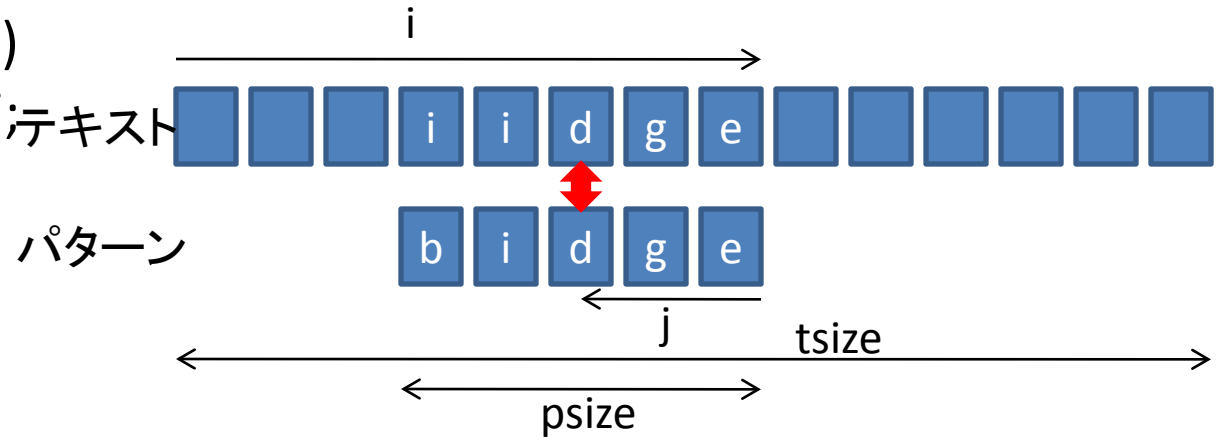
対応表作り:まずパターン長
パターンj番目の移動距離=パターン長-j-1
jが0からだから
パターンとテキストが一致中
パターンとテキストが不一致

結果の出力③}

探索の部分

- i:パターンの右端のテキスト中での位置
- J:パターン内の比較位置(左から数えて)

```
found=0; i=psize-1;
while (i<tsize && !found){
  found=1;
  for(j=0; j<psize; j++){
    if(text[i-j]==pattern[psize-1-j]) continue;
    else {
    found=0;
    if(skip[text[i-j]]-j > 1)
      i += skip[text[i-j]]-j;
    else i += 1;
    break; }}}
```



ホールスプールのアルゴリズム

- 探索の部分

```
found=0; i=psize-1;
while (i<tsize && !found){
  found=1;
  for(j=0; j<psize; j++){
    if(text[i-j]==pattern[psize-1-j]) continue;
    else {
    found=0;
    i += skip[text[i]];
    break; }}}
```

↓ボイヤー・ムーアで対応する部分はこう↓

```
if(skip[text[i-j]]-j > 1)
    i += skip[text[i-j]]-j;
else i += 1;
```

比較位置の移動量でなくパターン右端の移動量
逆行のチェックが不要

文字列照合の計算量

- パターン長M
- テキスト長N
- ボイヤー・ムーアのアルゴリズムも、ホールスプールのアルゴリズムも最悪の場合O(NM)