I111 アルゴリズムとデータ構造
第2回: アルゴリズムの基礎

北陸先端科学技術大学院大学
情報科学・融合科学分野　教授
池田　心
kokolo@jaist.ac.jp

2024-04-17 10:50-12:30

第2回の内容

- 良いアルゴリズム・悪いアルゴリズムの例
- 例1：株の最大売却益を求めるアルゴリズム
  - 素朴な方法
  - ちょっとした改良
  - さらなる改良: O(n^2) から O(n)へ
- 例2：$293^{10,000,000}$の下3桁を求めるアルゴリズム
  - 素朴な方法とその問題点
  - 10乗計算を利用した効率化
- これらの例は重要でない割にややこしいが，「へえ，そんなに効率化できるのか」と感じられればよい

第一回のTHでは素数列挙の例を示しました

アルゴリズム(algorithm)とは

- 計算機を用いて解ける問題に対する解法を抽象的に記述
  - どんな入力に対しても正しい解を返す
  - 必ず終了する

- プログラム: 計算機言語で記述したもの
  - 一見動いているように見えても,
    - 入力によっては正しい解を返さない
    - 入力によっては暴走，停止しない

Al-Khwarizmi

注）それもあって，資料ではいろんな言語の記法が混じってます

アルゴリズムの良い設計

- 設計技法を身につけ，よくあるパターンを利用する
- 計算時間, メモリ量の推定する
- アルゴリズムの正しさを検証・証明する

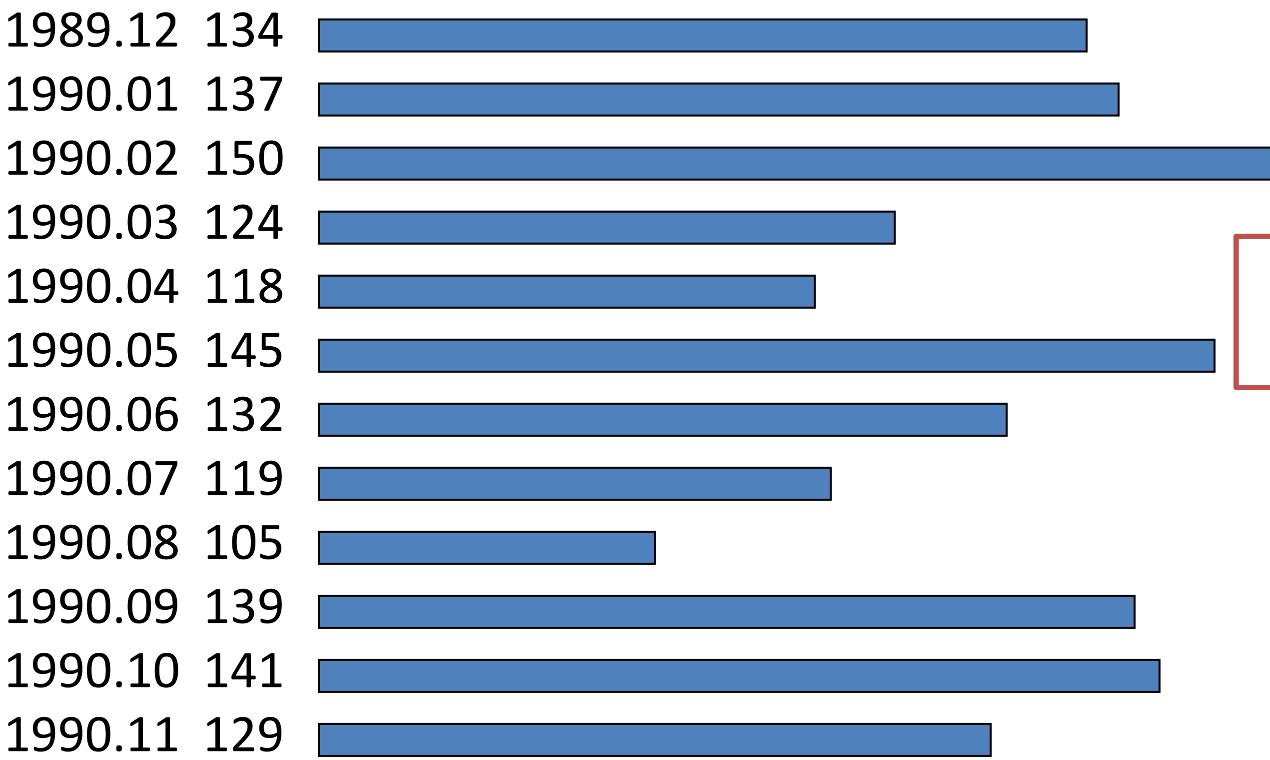
- アルゴリズムの悪い設計
  – 思いつき: アルゴリズム設計技法の知識の欠如
  – 作りっぱなし: アルゴリズムの動作の解析なし
  – とりあえず動かしてみよう・・・なんかバグった．なんか遅い．なんか多分動いてる・・・

いくつかの例で，効率の良い／悪いアルゴリズムを見ていく

例1：　最大売却益の求め方

例: 株で儲ける(最大売却益)

- 株を買って売ったときの収益の最大値は？



＊一回1株買って
一回1株売る

プチ演習：　まずは手作業で考えてみる

問題の定式化

- int sp[n]: 株価を蓄える配列 (nは適当な数)
- 時点 i に買って時点 j に売るとすると
  - 買値: sp[i]
  - 売値: sp[j]
  - 利益: sp[j] - sp[i]
- sp[j]-sp[i]の最大値
  max{sp[j] - sp[i] | 0<= i < j < n}
  を求める

~~~~~~ ここポイント　なぜ必要？
~~~~~~

アルゴリズムの大枠

- 方式A　買うタイミングを先に固定

Pascal風表記
（後ろに動くコードもあります）

```
for i=0 to n-2
  for j=i+1 to n-1
    利益sp[j]-sp[i]を求め，最大記録更新をチェック
```

← for (i=0; i<=n-2; i++) のこと

- 方式B　売るタイミングを先に固定

```
for j=1 to n-1
  for i=0 to j-1
    利益sp[j]-sp[i]を求め，最大記録更新をチェック
```

方式Aのアルゴリズム

- 以下のアルゴリズムは効率的か？

```
最大売却益(sp[],n){/*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最大値*/
  for i=0 to n-2
    for j=i+1 to n-1
      d = sp[j] – sp[i]; /*売却益*/
      if d > mxp then mxp = d;
                    /*最大売却益の更新*/
    endfor
  endfor
  return mxp;
}
```

Pascal風表記

← if (d>mxp) mxp =d; のこと

← for … { } の終わりのこと

● $n^2/2$ くらいの引き算や比較を行う

参考：C#版　動くプログラム

```
using static System.Console;
class Program{
    public static void Main(){
        int[] sp = new int[]{3,1,4,1,5,9,2,6,5,3,5};
        int n=sp.Length;  // 11

        int mxp = 0;
        for (int i=0; i<=n-2; i++) {
            for (int j=0; j<=n-1; j++) {
                int d = sp[j] - sp[i];
                if (d>mxp) mxp = d;
            }
        }
        WriteLine("max profit = "+mxp);
     }
}
```

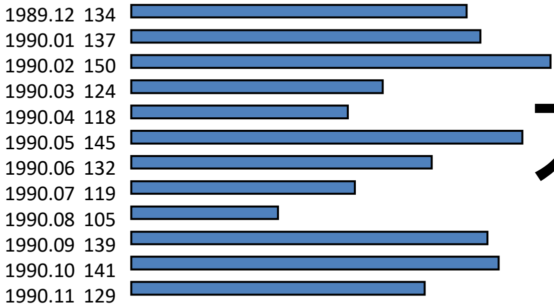
方式Aのアルゴリズム

- 以下のアルゴリズムは効率的か？

```
最大売却益(sp[],n){/*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最大値*/
  for i=0 to n-2
    for j=i+1 to n-1
      d = sp[j] – sp[i]; /*売却益*/
      if d > mxp then mxp = d;
                    /*最大売却益の更新*/
    endfor
  endfor
  return mxp;
}
```

【気づき】

iを固定すると、売却益が最大になるのは sp[j]が最大になるとき
➔ 毎度sp[j]-sp[i]を計算する必要はない



方式Aのアルゴリズム(改良版)

```
最大売却益(sp[],n){ /*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最大値*/
  for i=0 to n-2
    mxsp = sp[i];
    for j=i+1 to n-1
      if sp[j] > mxsp then mxsp = sp[j];
    endfor
    d = mxsp - sp[i];
    if d > mxp then mxp = d;
  endfor
  return mxp;
}
```

i以降の株価の最大値をmxspに

引き算がループの外側に

最大売却益の比較も外側に

● 引き算は n回くらいに減ったが，比較は．．．

アルゴリズムの大枠（P8再掲）

- 方式A　買うタイミングを先に固定

```
for i=0 to n-2
  for j=i+1 to n-1
    利益sp[j]-sp[i]を求め，最大記録更新をチェック
```

- 方式B　売るタイミングを先に固定

```
for j=1 to n-1
  for i=0 to j-1
    利益sp[j]-sp[i]を求め，最大記録更新をチェック
```

方式Bのアルゴリズム（素直版）

```
最大売却益(sp[],n){ /*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最大値*/
  for j=1 to n-1
    mnsp = sp[j];
    for i=0 to j-1
      if sp[i] < mnsp then mnsp = sp[i];
    endfor
    d = sp[j] - mnsp;
    if d > mxp then mxp = d;
  endfor
  return mxp;
}
```

株価のjまでの最安値をmnspに

● これだと P12のものと何も変わらない

アルゴリズムの効率

n^2に比例する程度の量であることを表す記法

- 繰り返し回数
  - 方式(A): ループの回数は O(n^2)

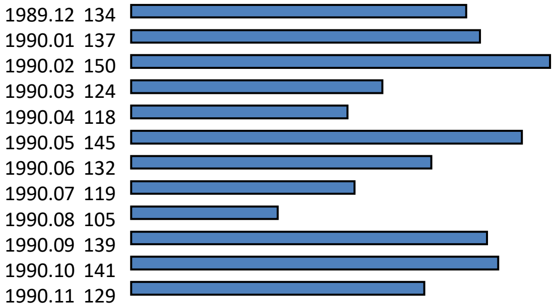
第4回で詳しく

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{n^2-n}{2} \leq n^2/2$$

  - 方式(B): ループの回数は O(n^2)

$$\sum_{j=1}^{n-1} \sum_{i=0}^{j-1} 1 = \sum_{j=1}^{n-1} j = \frac{n^2-n}{2} \leq n^2/2$$

Q. ループの回数は減らせない？



アルゴリズムの改善
ループの回数を減らす

- 2つ目のforループの中身を考える
  - A方式(P12):
    - MAX[i,n-1]を時点iから時点n-1までの最高値とする
    - MAX[1,n-1], MAX[2,n-1],…の順に計算
    - Q: 前回のMAX[i-1,n-1]を使ってMAX[i,n-1]を計算できる?

NO **調べる範囲が狭くなってるから**

  - B方式(P14):
    - MIN[0,j-1]を時点0から時点j-1までの最安値とする
    - MIN[0,0], MIN[0,1], …の順に計算
    - Q: 前回のMIN[0,j-1]を使ってMIN[0,j]を計算できる?

YES! `MIN[0,j] = min(MIN[0,j-1],sp[j])`

方式Bのアルゴリズム(P14再掲)

```
最大売却益(sp[],n){ /*sp[0] sp[n-1]*/
  mxp=0; /*利益の最大
  for j=1 to n-1
    mnsp = sp[j];
    for i=0 to j-1
      if sp[i] < mnsp then mnsp = sp[i];
    endfor
    d = sp[j] - mnsp;
    if d > mxp then mxp = d;
  endfor
  return mxp;
}
```

- j=あるkのとき: mnspはsp[0]からsp[k-1]までの中の最安値
- 次，j=k+1のとき: mnspはsp[0]からsp[k]までの中の最安値

これまでの最安値msfをとっておいて次はmsfと先月価格sp[k]の最小値をとればそれが最安値になる

効率の良いアルゴリズム

- O(*n*) 時間で計算するアルゴリズム

質問：そもそも,
sp[]の最大値－sp[]の最小値
では間違いなんでしょうか？

```
最大売却益(sp[],n){ /*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最高値*/
  msf = sp[0]; /*これまでの最安値*/
  for j=1 to n-1
    d = sp[j] - msf;
    if d > mxp then mxp = d;
    if sp[j] < msf then msf = sp[j];
  endfor
  return mxp;
}
```

最安値更新作業

魔法みたい！
O(n)と
$O(n^2)$ は泣けるほど違う

演習：P10のコードをこの方式に書き換えよ

いくつかの例で，効率の良い／悪いアルゴリズムを見ていく

例2：冪乗の計算

$293^{10,000,000}$

冪乗の計算

- 問い: $293^{10,000,000}$の下位3桁の数字を求めよ
- 方法1: 素朴な解法
  293を10,000,000回掛けて下から3桁出力

```
int a,b;
a=1;
for(i=1;i<=10000000;i=i+1)
  a=a*293;
b=a%1000; /*下三桁*/
printf(“answer=%d”,b);
```

出題者

徳山豪教授
（東北大学）

- この方法の問題点は？

冪乗の計算
素朴な解法の問題点

- (a) パソコンでは扱えないかもしれない
  - 32bitで整数を表すとすると最大値は $2^{31}-1 = 2,147,483,647$
  - 293を10,000,000回掛けると桁数は $10,000,000 \cdot \log_{10} 293 \geqq 24,668,676$
  - 各桁を配列に入れるような方法もあるが面倒
- (b) 非常に遅い
  - 単純に掛け算が10,000,000回

冪乗の計算
(a) 桁数爆発を抑える

- 必要なのは下三桁であることに注目
  - 293*293=85,849
  - 85,849*293
= (85***1000**+849) * 293
= **85*1000*293** + 849*293

下三桁には影響しない

- 毎回の計算で下三桁だけ求めればよい
  - a=a*293 ➔ a=(a*293)%1000

冪乗の計算
改良されたプログラム

```
int a;
a=1;
for(i=1;i<=10000000;i=i+1)
  a = (a*293) % 1000;
printf(“answer=%d”,a);
```

これなら動く！

- (b) 依然として10,000,000回の乗算*と剰余%が必要.
  画期的な高速化は?

冪乗の高速化: 10乗するコストは？

- 方法1: 単純に10回掛ける
- 方法2: 2進数展開の利用
  - $10_{(10)} = 1010_{(2)} = 8 + 2$
  - 2乗を繰り返して x^2, x^4, x^8 を計算して $x^{10} = x^8 \times x^2$ とすると4回の乗算で済む
    - 例: $3^{\underline{2}} = 9$, $3^4 = 9^{\underline{2}} = 81$, $3^8 = 81^{\underline{2}} = 6,561$, $3^{10} = 3^8 \times 3^2 = 6,561 \underline{\times} 9 = 59,049$

冪乗の高速化: 10乗を使って計算

- 関数 power10(x)を以下のように定義する:

```
int power10(int x){
  int x2,x4,x8;
  x2=(x*x)%1000; x4=(x2*x2)%1000;
  x8=(x4*x4)%1000;
  return (x2*x8)%1000;
}
```

- $293^{10,000,000}$は以下のように計算できる:
- 10乗して10乗して･･･を7回繰り返す

```
int i,x; x=293;
for(i=0;i<7;i=i+1) x=power10(x);
printf("answer=%d",x);
```

4×7= 28回の乗算で済む！

10乗を使って計算（動くプログラム）

```
using static System.Console;
class Program{
    static void Main(string[] args){
        int x=293;
        for (int i=1; i<=7; i++) {
            x=power10(x);
        }
        WriteLine("(293^10000000)%1000="+x);
    }

    static int power10(int x) {
        int x2 = (x*x)%1000;
        int x4 = (x2*x2)%1000;
        int x8 = (x4*x4)%1000;
        return (x2*x8)%1000;
    }
}
```

2進数展開を用いた$293^{10,000,000}$の計算

- 10乗を7回使う方法は，一般のm^nに使えない
- 2進数展開なら，一般に適用できる（P32-）
- $10,000,000_{(10)} = 100110001001011010000000_{(2)}$
  - $x^{10000000}=x^{8388608}$(2の23乗) $\cdot x^{1048576} \cdot x^{524288} \cdot x^{32768} \cdot x^{4096} \cdot x^{1024} \cdot x^{512} \cdot x^{128}$
  - $x^2, x^4, x^8, x^{16}, \ldots x^{8388608}$ の計算は23回の乗算
  - 8つの値の掛け算に7回
  - 合計で30回の乗算
- power10を用いたとき（28回）よりは効率が悪い

冪乗計算: まとめ

- 方法0: 素朴な方法
  プログラムは動かない
- 方法1: 下3ケタ以外を無視すると. . .
  動くけど, 10,000,000回の乗算
- 方法2: 10乗を計算する手続きを利用
  全体で28回の乗算と剰余で済む
- 方法3: 2進数展開を使う
  一般的に動くように記述可能（30回乗算）
- （方法4: オイラーの定理を利用: P36-）

方法によって大きな違いが生じる
アルゴリズムって大事

演習（過去問）

- a^4から a^{20}くらいまで，一番掛け算の回数が少なくなる方法を考えてみよう
- 例外的なやつがあるはず

以下付録

おまけ

- では x^k を計算するときの乗算の最小回数はいくつなのか？
  - $x^2, x^4, x^8, x^{16}, \ldots$ を計算する方法で、$O(\log k)$という上界（それだけやれば少なくとも十分）は得られる
  - この方法で得られる回数が最小でない例：$k=15$
  - 興味のある人は以下を調べるとよい：

本当？

The Art of Computer Programming, D. Knuth, Vol. 2, Chapter. 4.6.3. （邦訳もあり）

この本はコンピュータサイエンス業界のバイブル。
Knuthは業界の巨人。この本を書くためにKnuthは$\TeX$を作った。

【高度な内容】

2進数展開のプログラム

```
int n,k,bit[100];
n=/*some integer*/;
k=0;
do {
  bit[k]=n%2; k=k+1;
  n=n/2;
} while (n>0);
while (k>0) {
  k=k-1;
  print("%d",bit[k]);
}
```

n=49の場合

```
  bit[0]=1  n=24
  bit[1]=0  n=12
  bit[2]=0  n=6
  bit[3]=0  n=3
  bit[4]=1  n=1
  bit[5]=1  n=0, k=6
```

出力: 110001

【高度な内容】

2進数展開を用いた冪乗の計算

- 整数 n の2進数表現を
  $n = b_k 2^k + b_{k-1} 2^{k-1} + \cdots + b_1 2^1 + b_0 2^0$
  とするとき

$$x^n = x^{b_k 2^k} \times x^{b_{k-1} 2^{k-1}} \times \cdots \times x^{b_1 2^1} \times x^{b_0 2^0}$$

よって，$b_i = 1$ のところの積をとればよい

$$x^{49} = x \times x^{16} \times x^{32}$$

【高度な内容】

2進数展開を用いた冪乗の計算: プログラム

```
p=x; t=1;

if(n%2==1) t=t*p;
n=n/2;
do {
    p=p*p;
   if(n%2==1)
      t=t*p;
   n=n/2;
} while (n > 0)
printf(“%d”,t);
```

ビットが1の所を乗算

- 例: $n = 49_{(10)} = 110001_{(2)}$

n	p	t	乗算回数計
49/2=24	x	x	1
24/2=12	x^2	↑	2
12/2=6	x^4	↑	3
6/2=3	x^8	↑	4
3/2=1	x^{16}	x^{17}	6
1/2=0	x^{32}	x^{49}	8

【高度な内容】

2進数展開を用いた冪乗の計算: プログラム

```
p=x; t=1;

if(n%2==1) t=t*p;
n=n/2;
do {
  p=p*p;
  if(n%2==1)
    t=t*p;
  n=n/2;
} while (n > 0)
printf("%d",t);
```

ビットが1の所を乗算

- Q: 繰り返し回数は？
- A: $\lfloor \log_2 n \rfloor$ 回
  - $2^{k-1} < n \leqq 2^k$ のとき k 回目の繰り返しで $n=0$
  - $\lfloor \ \rfloor$ は切り下げ

【高度な内容】

冪乗の高速化: 10乗の計算

- 計算の様子
  - 293^{10} mod 1000 = 249
  - 293^{100} mod 1000 = 1
  - $293^{1,000}$ mod 1000 = 1
  - $293^{10,000}$ mod 1000 = 1
  - $293^{100,000}$ mod 1000 = 1
  - $293^{1,000,000}$ mod 1000 = 1
  - $293^{10,000,000}$ mod 1000 = 1

【高度な内容】

オイラーの定理

- *n* と互いに素な *x* について

$$x^{\phi(n)} \bmod n = 1$$

- Φ(*n*) はオイラーのファイ関数と呼ばれ、*n* 以下で *n* と互いに素な数の個数を表す
  - $n = \prod_{i=1}^{d} p_i^{k_i}$ と素因数分解できるとき

$$\phi(n) = n \prod_{i=1}^{d} \left(1 - \frac{1}{p_k}\right)$$

- チェック: 二つの自然数が互いに素とは？

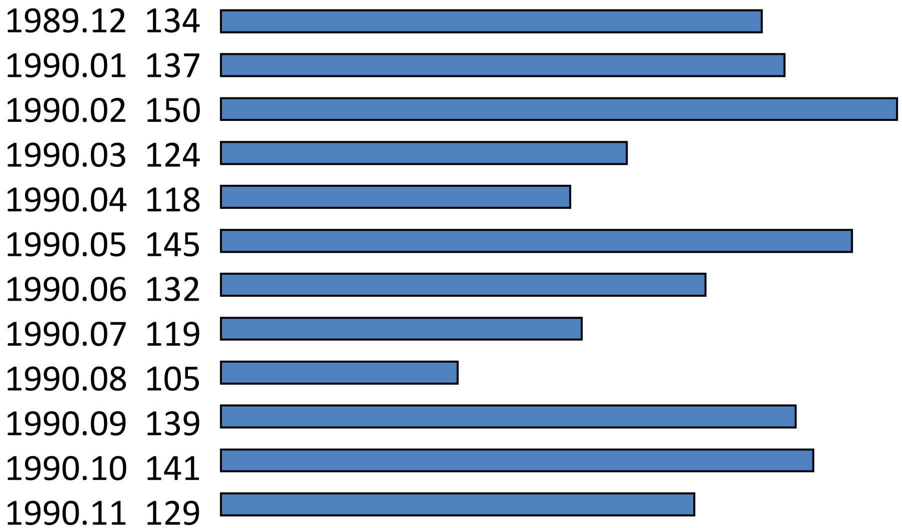
【高度な内容】

オイラーの定理を使った $293^{10,000,000} \bmod 1000$ 計算の高速化

- $\Phi(1000) = \Phi(2^3 \times 5^3) = 1000 \times 1/2 \times 4/5 = 400$
  - $\Phi(n)$ は n の素因数分解が出来れば計算できる
- 293 と 1000 は互いに素である
  $\Rightarrow 293^{\Phi(1000)} \bmod 1000 = 293^{400} \bmod 1000 = 1$

- $293^{10,000,000} \bmod 1000$
  $= (293^{400} \bmod 1000)^{25,000} \bmod 1000$
  $= 1^{25,000} \bmod 1000$
  $= 1$

I111-02 ワークシート



- 利益の最大値は【　　　　】万円
- 求め方は____
- その方法は実装できそう？ Yes/No
- その方法はどんな場合も？　OK/NG

a^4から a^{20}くらいまで，一番掛け算の回数が少なくなる方法を考えてみよう

$a^2 = a \times a$; $a^4 = a^2 \times a^2$　（ a^4 は2回）

$a^3 = a \times a \times a$;　$a^9 = a^3 \times a^3 \times a^3$

$a^2 = a \times a$;　$a^4 = a^2 \times a^2$;　$a^8 = a^4 \times a^4$;　$a^9 = a^8 \times a$（a^9は2つの方法で4回）