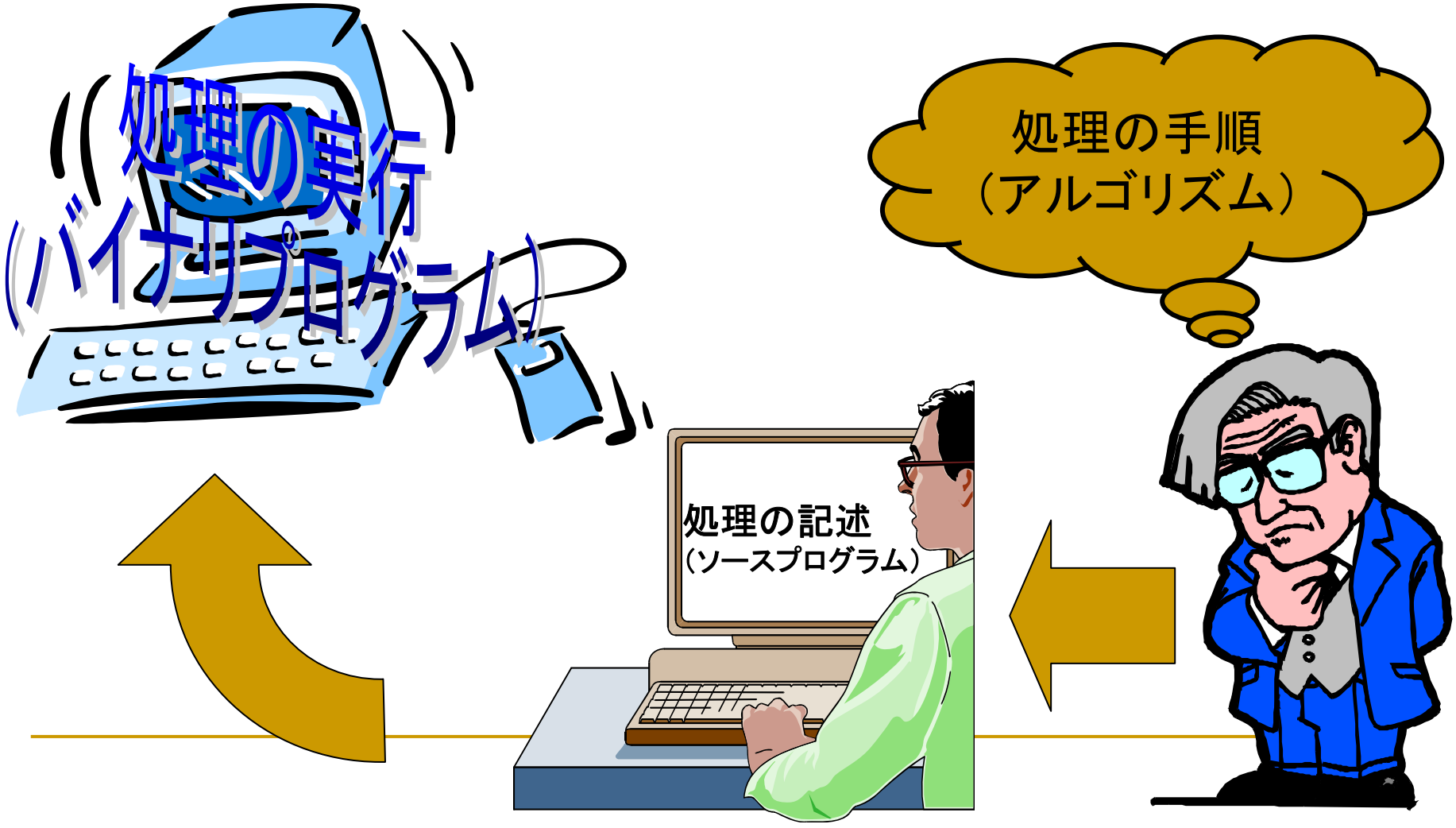
コンピュータ概論　第11回

さまざまなアルゴリズムと
プログラム

プログラムとアルゴリズム



問題解決とアルゴリズム

- 問題解決という目的を達成するにはアルゴリズムを作らなければならない
  - アルゴリズムの良し悪しで仕事の効率が変わる
    - 要領の良い店員と要領の悪いアルバイト
    - ベテランと初心者
  - 仕事の効率＝処理速度
    - コンピュータはより高速に処理を行うためのもの
    - 効率は出来るだけ良くしなければならない

単なる数値計算でも…

- 式の書き方を工夫する
  - 数式をそのまま

    $y=x^3-4x^2+3x-5$ ⇒ y=x*x*x−4*x*x+3*x−5;
  - 工夫すると…

    y=x*x*x−4*x*x+3*x−5 ⇒ y=x*(x*(x−4)+3)−5;
  - 積の回数が減る⇒速度向上、誤差減少

アルゴリズムの定石

- 定石
  - 囲碁で、昔から研究されてきて最善とされる、きまった石の打ち方
- アルゴリズムの定石
  - 定番問題ごとに最善とされる、きまったアルゴリズムの形

漸化式を扱うなら、

- 数列の漸化式は、数列の次の要素を求めるアルゴリズムでもある

  例）$a_0=4$，$a_{n+1}= n \cdot \log(a_n)$

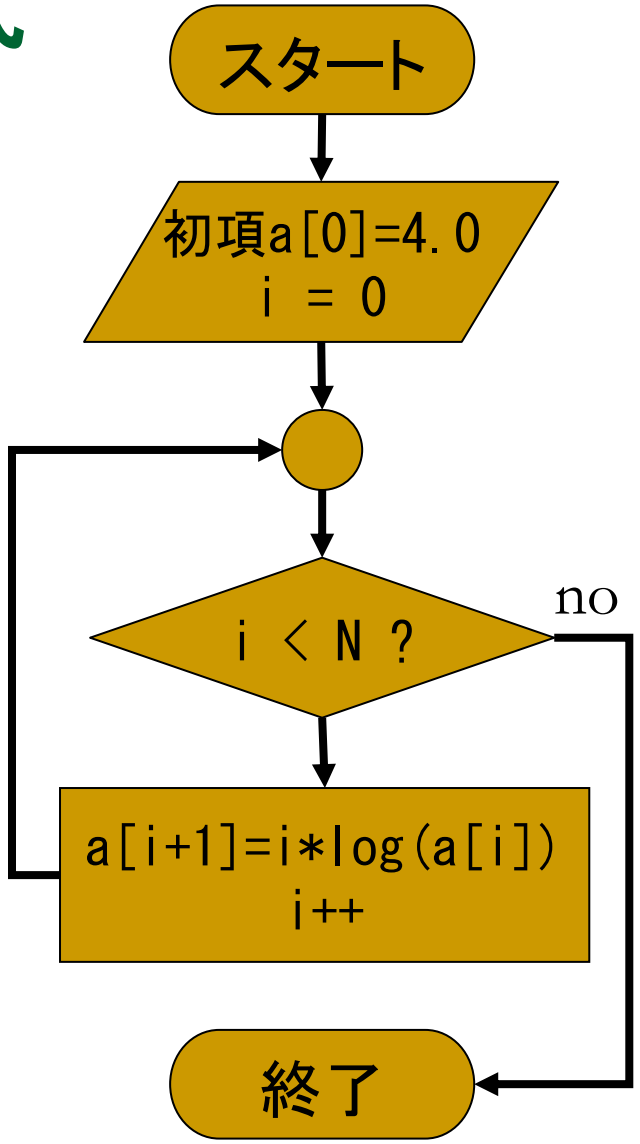
- 数列は配列で表すのが適している
  - 繰り返し処理も必須

```
a[0]=4.0;
a[n+1] = n * log(a[n]);
```

- 配列の添え字（要素番号）は言語によって異なる
  - Cは0から，PascalやFORTRANは1から始まる

フローチャート：漸化式

- 初項を決めれば後は繰り返しで求まる
  - 第N項までを求める例

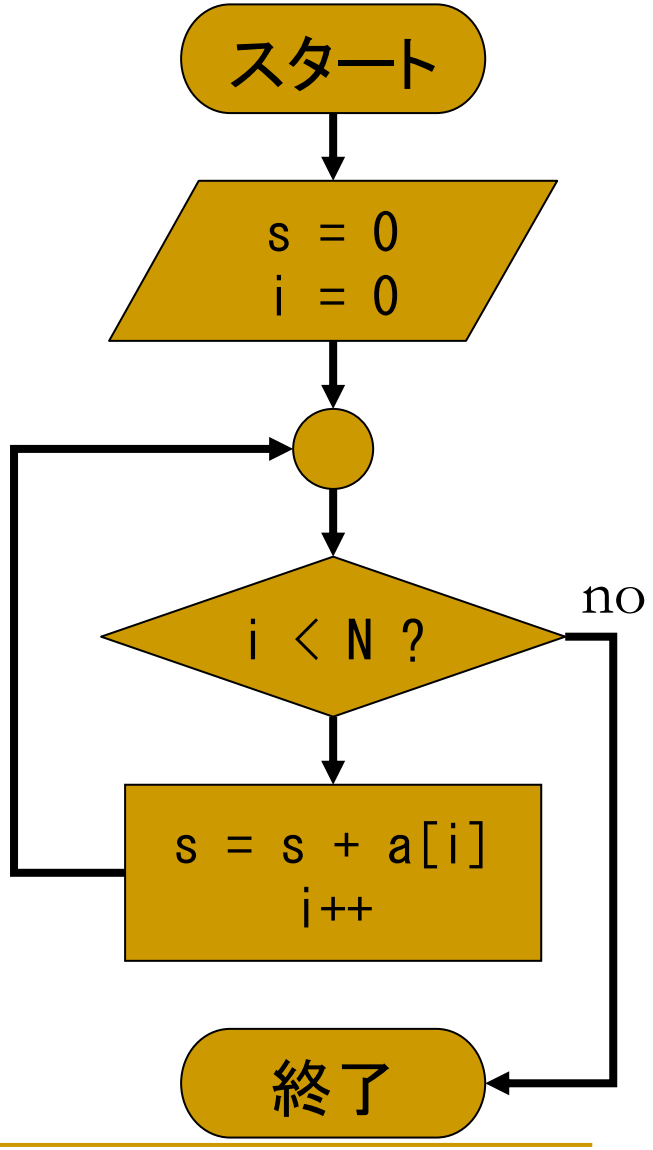


数列の総和

- 数列の総和を求める
  - 数列全てを一度に足すのは非現実的
    - 記述が大変（数列が100項あったら!!）
  - 要素を1つずつ足し合わせていく
    - 繰り返し処理が必要なので練習問題に適している
    - 一時変数の活用（その時点の合計を代入する）
- ちなみに、
  - 数列の積を求めることも簡単
  - 積の場合は、変数の型のビット長に注意
    - オーバーフローするとNG
  - 一時変数の初期値は1（総和のときは0）

フローチャート：数列の総和

- 総和を保持する変数の初期値が重要
  - 第N項までの総和を求める例
  - 積を求めるなら初期値は1
    - 例えば階乗n!を求める

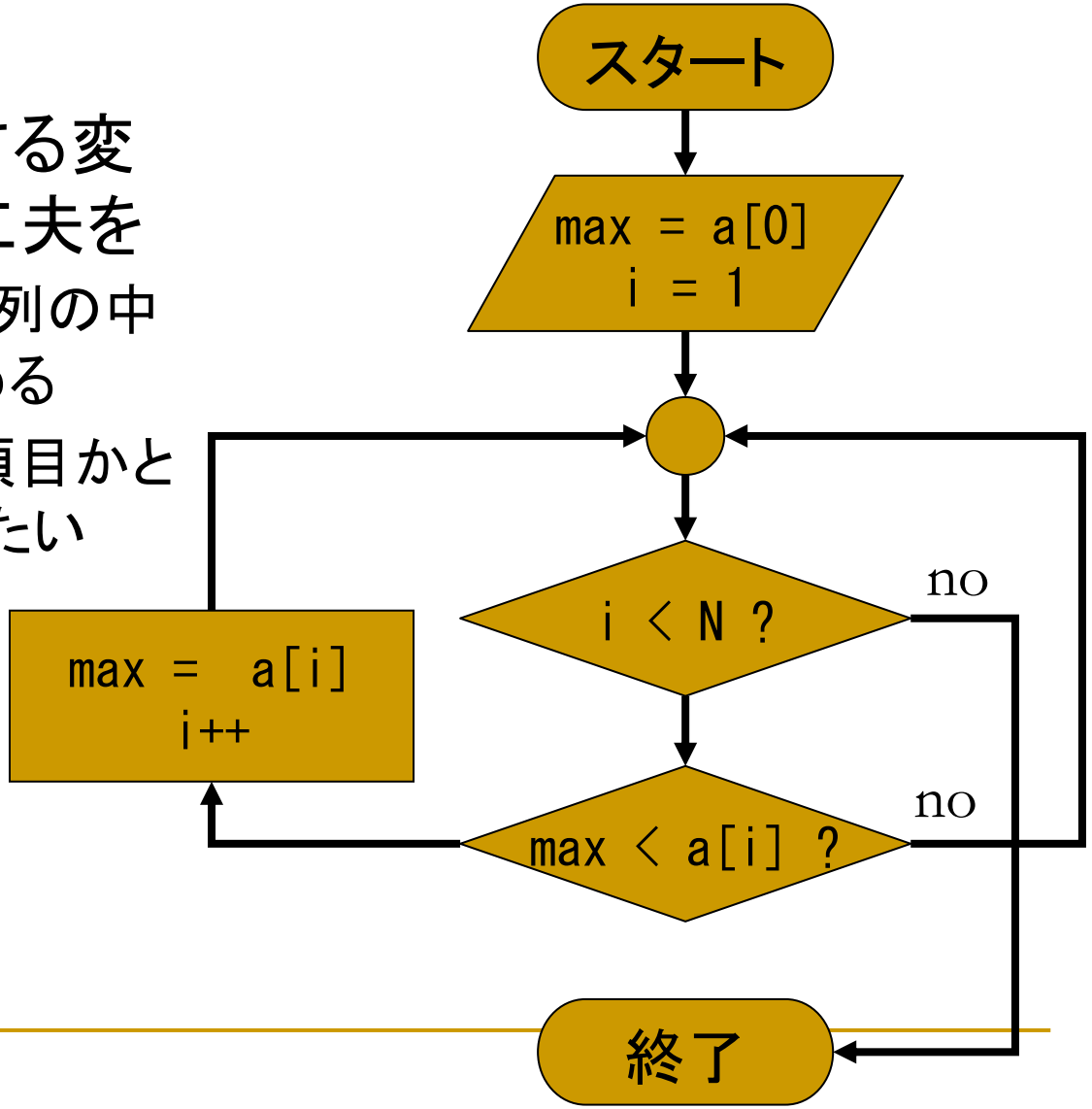


最大値を求める

- 配列要素の中から最大のものを見つける
  - コンピュータは同時に2つの値を比較することしか出来ない（比較演算子）
    - 天秤で比べているのと同じ
  - 条件判断と繰り返し処理の入門向け問題
    - 1つずつ比較する
    - 一時変数の活用（その時点の最大値を代入する）
    - 最大値を求めたら、最小値も
  - 並べ替え（ソート）問題への導入
    - 2番目に大きい値は？

フローチャート：最大値を求める

- 最大値を保持する変数の初期値に工夫を
  - 第N項までの数列の中の最大値を求める
  - 最大値は第何項目かという問題も考えたい



アルゴリズム例：ソート

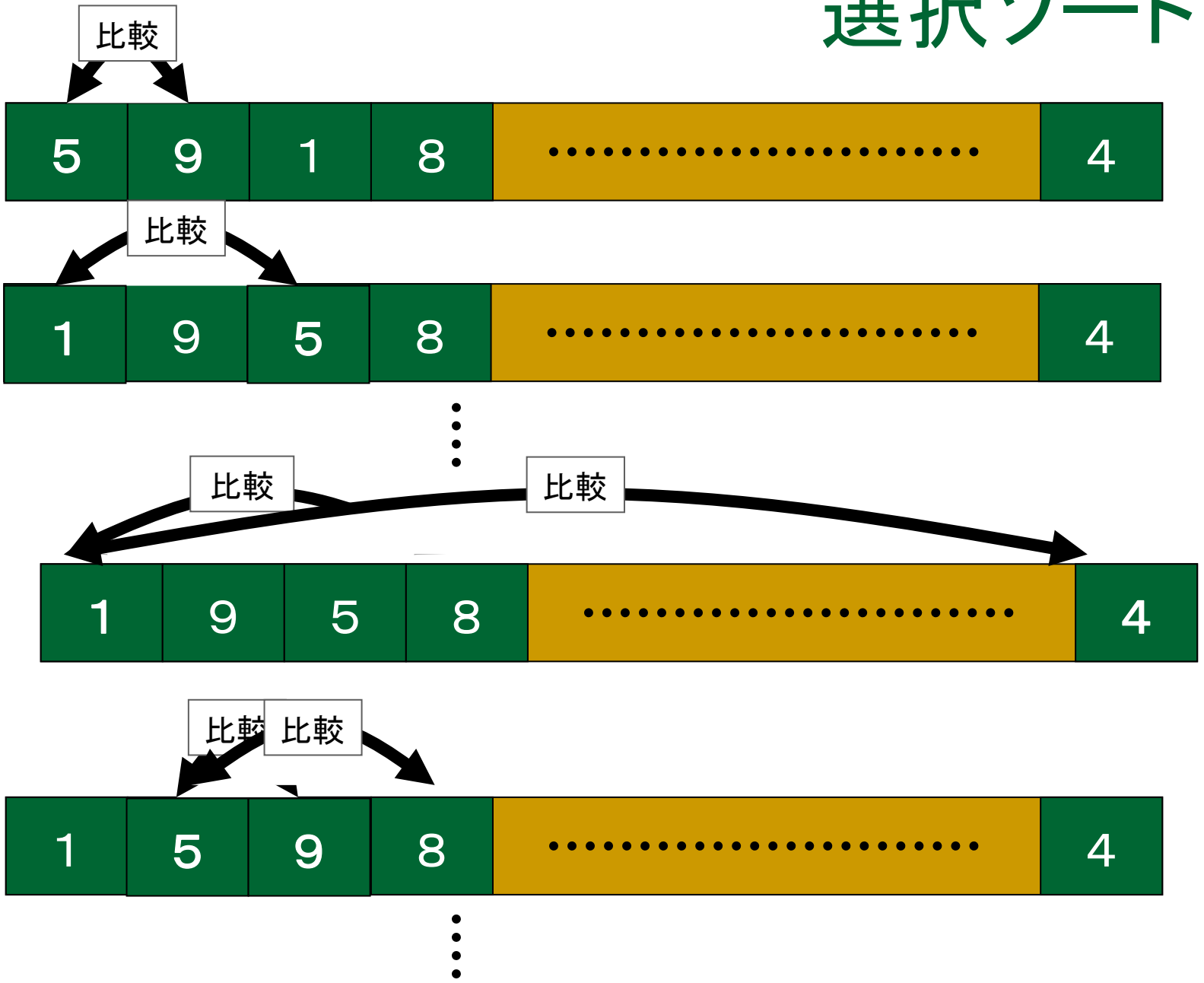
- 配列の要素を、順番どおりに並べ替える
- アルゴリズム勉強の定番
  - 稚拙な方法から高度な方法まで数多く知られている
  - 自分が考えた方法の効率がすぐわかる。
    - 選択ソート（馬鹿ソート）
    - 挿入ソート
    - シェルソート
    - バブルソート
    - クイックソート　　などなど

選択ソート（小さい順）

初心者レベル（要素N+1ならN^{2}回の比較・入替）

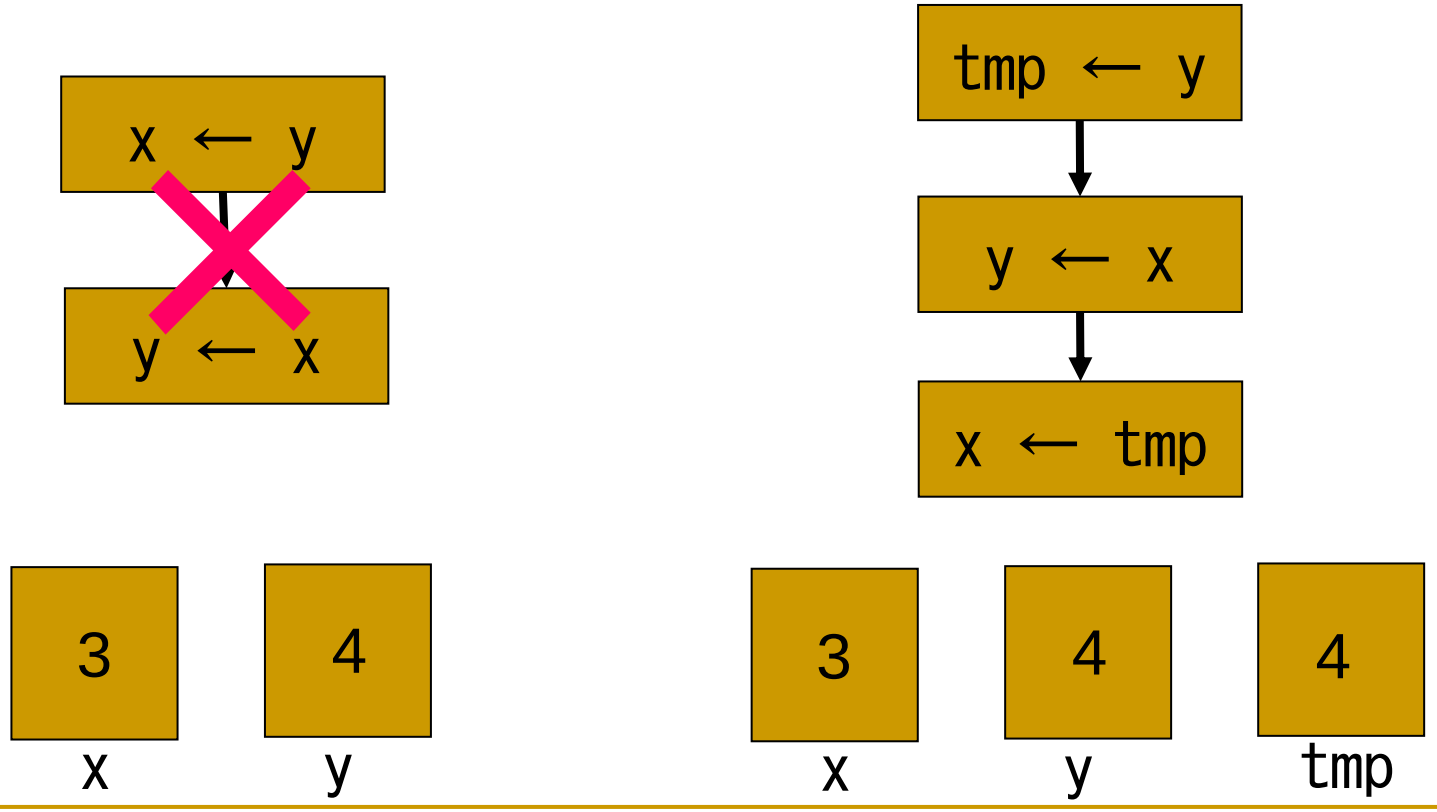
- 1番目の要素と2番目以降を順番に比較
- 1番目の要素よりも小さい要素があれば、その要素と1番目の要素を入れ替え
- 最後の要素まで進んだら、2番目の要素と3番目以降を順番に比較
- 同様にして、最後の要素まで達したら終了

選択ソート



おまけ:2変数のスワッピング

- 変数xと変数yに代入されている値の入れ替え（スワップ）をするにはどうすればよいか？



選択ソートプログラム

```
#include<stdio.h>
main(void)
{
  int i, j, m, t, x[10];
  printf("小さい順に並べ替えます。¥n");
  for (i = 0; i < 10; i++){
    printf("%d番目の数=", i + 1);
    scanf("%d", &(x[i]));
  }
  for (i = 0; i < 9; i++){
    for(j = i + 1; j < 10; j++){
      if (x[i] > x[j]){
        t = x[i];
        x[i] = x[j];
        x[j] = t;
      }
    }
  }
  for (i = 0; i < 10; i++){
    printf("%d番目の数：%d¥n", i + 1, x[i]);
  }
}
```

バブルソート（小さい順）

中級レベル（要素N+1ならN^2回の比較・入替）

- 1番目の要素と2番目を比較して、1番目の要素よりも小さければ入れ替え
- 次に2番目の要素と3番目を比較して、2番目の要素よりも小さければ入れ替え
- 同様にして、最後の要素まで達したら、再び1番目からはじめる
- 最後まで入れ替えが無ければ終了

バブルソート

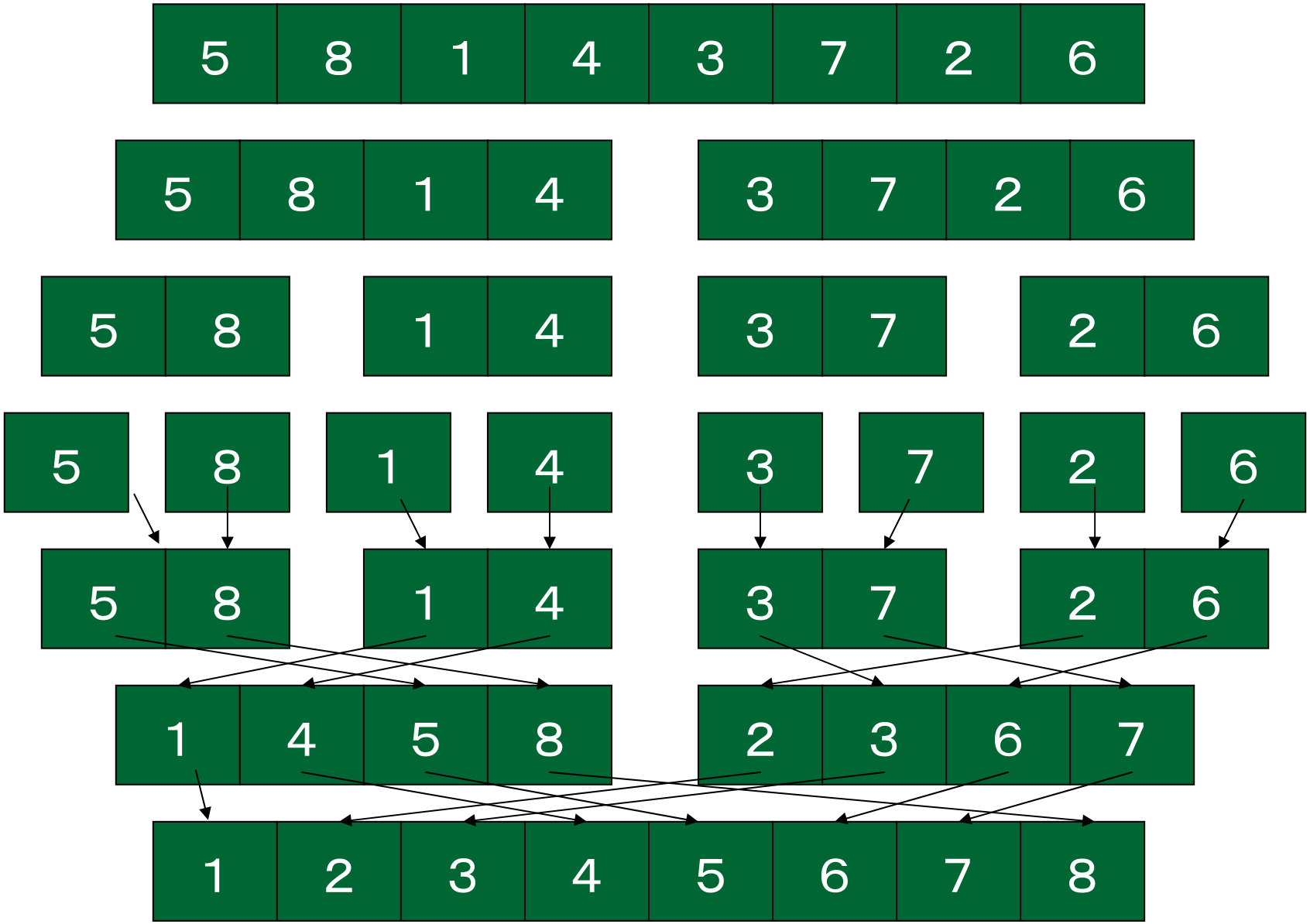
バブルソートプログラム

```
#include<stdio.h>
main(void)
{
  int i, j, m, t, x[10];
  printf("小さい順に並べ替えます。¥n");
  for (i = 0; i < 10; i++){
    printf("%d番目の数=", i + 1);
    scanf("%d", &(x[i]));
  }
  do{
    i = 0;
    for(j = 0; j < 9; j++){
      if (x[j] > x[j + 1]){
        t = x[j];
        x[j] = x[j + 1];
        x[j + 1] = t;
        i++;
      }
    }
  } while (i > 0);
  for (i = 0; i < 10; i++){
    printf("%d番目の数：%d¥n", i + 1, x[i]);
  }
}
```

マージソート（小さい順）

- 上級レベル
  - 概略
    - 配列を半分ずつにしていき、元に戻す時に、順番どおりに並べ替えていく
  - 分割統治法
    - 問題を分割して、小さい簡単な問題にしていく
    - アルゴリズムの効率化では常套手段
  - 要素N+1なら最速でN･logN回の分解・比較
    - 要素数が2のM乗個なら最速性能

マージソート


5 8 1 4 3 7 2 6
5 8 1 4
3 7 2 6
5 8
1 4
3 7
2 6
5
8
1
4
3
7
2
6
5 8
1 4
3 7
2 6
1 4 5 8
2 3 6 7
1 2 3 4 5 6 7 8

他にもいろいろ

- さまざまな定番問題に定石アルゴリズム
  - ニュートン法（方程式の解）
  - シンプソン法（積分）
  - ナップザックアルゴリズム（最適な割り当て）

  などなど
- 既存のアルゴリズムを活用する利点
  - 手間が省けて動作も保障される
  - 効率的で高速な処理が可能
- ただし，勉強しないとダメ

例）30個の整数の最大値を求める

```
#include<stdio.h>
main(void){
    int i, m, x[30];
    printf("最大値を求めます。¥n");
    for (i = 0; i < 30; i++){
        printf("%d番目の数=", i + 1);
        scanf("%d", &(x[i]));
    }
    m = 0;
    for (i = 1; i < 30; i++){
        if (x[i] > x[m]) m = i;
    }
    printf("%d番目の数が最大値%d¥n", m + 1, x[m]);
}
```

入力データ(30個の整数)

5	37
49	47
-9	1
1	-45
38	-9
-56	-2
2	3
6	12
23	7
-97	21
12	-33
47	0
6	61
10	-4
-28	6

プログラム実行における問題点

- 実行するたびにいちいちデータを入力しなければならない
- ソートなどのプログラムでは入力するデータの数が多くで面倒
- プログラムの動作結果を保存しておけない
- プログラムの動作結果の表示が多いと、表示がウィンドウに収まりきらない

リダイレクト

- キーボードから入力するデータを、事前に作成しておいて、自動的に入力を行なう
  - いちいち手入力しなくても良い
  - 毎回同じデータを入力できる
- プログラムの動作結果の表示を、そのままファイルとして保存する
  - 結果をとっておくことができる

リダイレクトが必要な例

- 30個の整数の中から最大の数を求める
  - 毎回30個のデータを手入力するのは大変
  - 入力をリダイレクトすれば何度でも実行できる
- フィボナッチ数列:$a_n = a_{n-1} + a_{n-2}$を第100項目まで求める
  - 表示が流れていってしまい、はじめのほうの項を見ることが出来ない
  - 出力をリダイレクトすれば後で見られる

入力リダイレクト

Z:¥compsys> 実行コマンド < 入力データファイル

- 実行コマンドは、プログラムの名前
- 入力データファイルは、手入力する場合と同じようにエディタで作成する
  - 入力する文字や数字をそのまま記述
  - Enterを押すところも同じようにEnterを押す
- 手入力の場合と異なり、入力したデータは画面に表示されない

出力リダイレクト

Z:¥compsys> 実行コマンド > 出力ファイル

- 実行コマンドは、プログラムの名前
- 実行したときに画面に表示される全ての文字が、出力ファイルの中に保存される
  - 実行中は画面には何も表示されない
  - 入力が必要なプログラムの場合は、それだけ表示されるが、その時の入力は出力ファイルには入らない
  - 表示を促すメッセージも表示されないので、手入力を伴うプログラムには向かない

リダイレクト応用

Z:¥compsys> 実行コマンド < 入力F > 出力F

- 入力F（ファイル）からデータが入力されて、結果が出力Fに入る
  - 入力データも実行結果も画面には表示されない
    - 次のプロンプトが表示された時点で実行終了
  - 出力ファイルには入力データは入らない
    - リダイレクトを使わない場合とはこの点が異なる

おさらい

- さまざまな定番プログラムと定石アルゴリズム
  - 数列の扱い
  - 最大値・最小値
  - ソート
- リダイレクト
  - 入力リダイレクト・出力リダイレクト

課題

- 30個の整数を入力データファイルに用意して、その中から2番目に大きい整数を求めるプログラムを作成せよ
  - 入力データは最大値を求める例題「30個の整数の最大値を求める」プログラムと同じものを使用する
  - 表示メッセージは全て独自のものに書き換える
  - 入力リダイレクトで実行する
  - 出力結果は出力リダイレクトで保存する
    - 保存した内容（出力結果）をプログラムにコメントとして書き加えてから提出
    - 自分のプログラムの出力結果を使うこと（チェック時に一致しないと再提出）

今回提出するときの注意点

```
main(){
   プログラムの中身
}
/*出力ファイル
a[0] = 5
a[1] = 49

   ⋮

a[24] = 2
最小値は...
*/
```

出力データファイルの中身を
エディタで読み込み、コピーして、
プログラムの後ろに貼り付ける
（前後のコメント記号を忘れずに）