プログラミング基礎
第9回

藤江 真也

2021年6月18日

準　備

- ファイル（0618.tgz）をダウンロード

```
$ wget http://sites.fujielab.org/ip/files/0618.tgz
```

- ダウンロードしたファイルを展開

```
$ tar zxvf 0618.tgz
```

- 展開されたディレクトリに移動

```
$ cd 0618
```

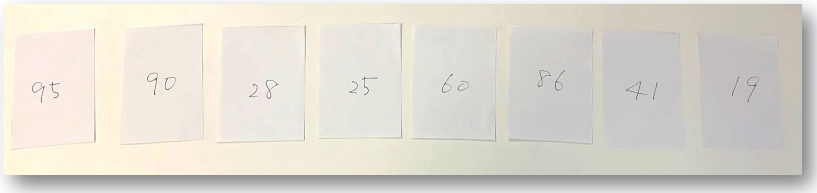
- random.cなどのファイルがあることを確認

```
$ ls
```


並べ替えの実習

準　備

- レポート用紙やA4の印刷用紙を8つ折りにして切り離し，小さいカード状の紙を8枚用意する
- `random.c`をコンパイルして実行し，出て来た8つの数字を1枚にひとつずつ書き出す



実習内容

- 問題
  - 数字が昇順（小さい順）になるようにカードを並べ替える
- ルール
  - 基本的に**すべてのカードを伏せて**作業する
  - 1度に**開けるカードは2枚**までとする
  - 過去に開いたカードの**数字を記憶してはいけない**
  - 開いている2枚のカードの**大小比較**とカードの**入れ替え**のみが可能
- 目標
  - **できるだけ手数を少なくする**
    - 大小比較，カードの入れ替え回数が少ないほど良い

ルールの詳細

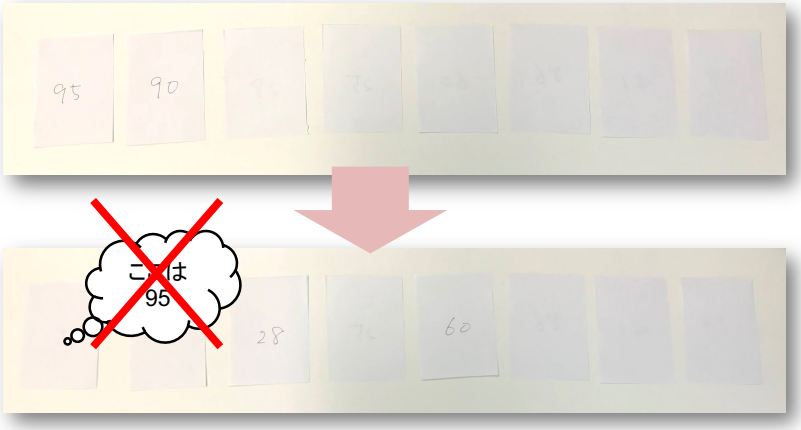
- 基本的にすべてのカードを伏せて作業する
  - 伏せたカードの数字は読めないこととする（薄く見えていたとしても）

ルールの詳細

- 1度に開けるカードは2枚までとする
  - どこの2枚を開いてもいいが，3枚以上は同時に開けないとする

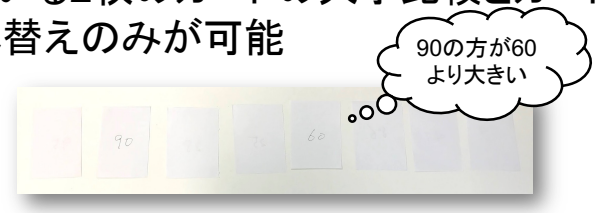
ルールの詳細

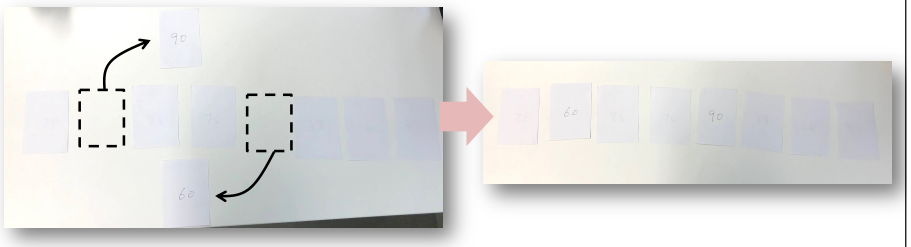
- 過去に開いたカードの数字を記憶してはいけない



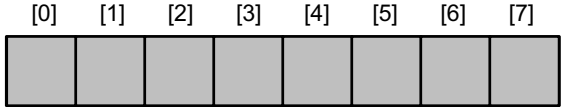
ルール詳細

- 開いている2枚のカードの大小比較とカードの入れ替えのみが可能

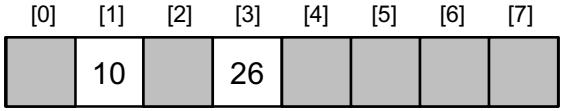




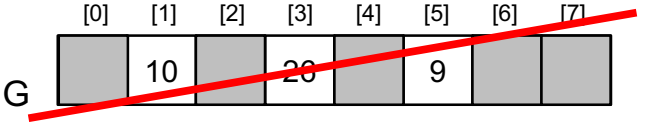
基本状態
（伏せておく）



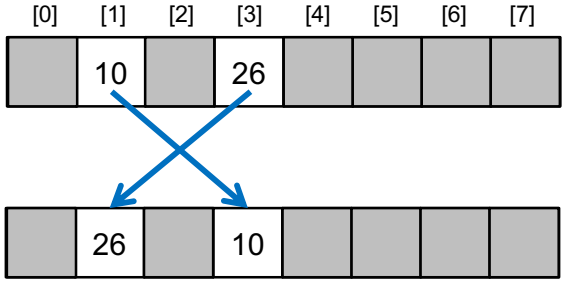
2枚まで
開いてOK



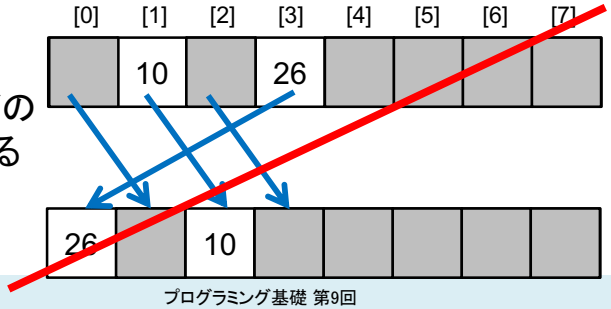
3枚以上
開くのはNG



2枚の
入れ替えは
OK



ほかのカードの
順番を変える
操作はNG



実習内容（再掲）

- 問題
  - 数字が昇順（小さい順）になるようにカードを並べ替える
- ルール
  - 基本的に**すべてのカードを伏せて**作業する
  - 1度に**開けるカードは2枚**までとする
  - 過去に開いたカードの**数字を記憶してはいけない**
  - 開いている2枚のカードの**大小比較**とカードの**入れ替え**のみが可能
- 目標
  - **できるだけ手数を少なくする**
    - 大小比較，カードの入れ替え回数が少ないほど良い

やってみよう！

説　明

ソート

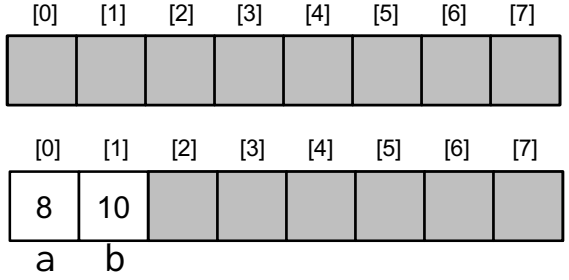
- 8個の数字の例

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
初期状態	8	10	2	26	6	9	14	18

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
期待される結果	2	6	8	9	10	14	18	26

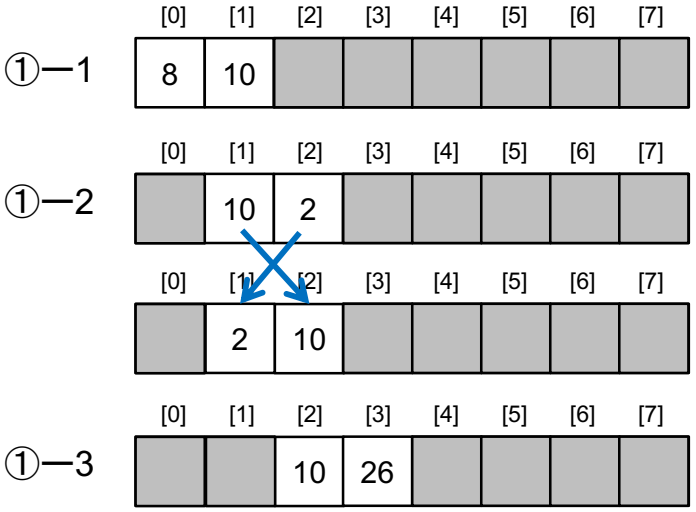
ソート

- **1度に見れるのは二つの数字**だけ
- できるのは**大小チェック**と**入れ替え**するかだけ



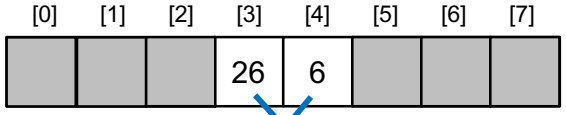
もし a > b なら入れ替える

ソートの例

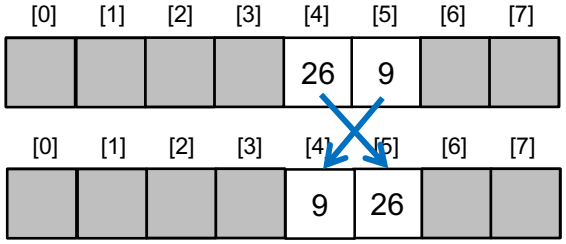


ソートの例

①—4

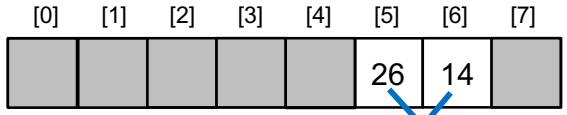


①—5

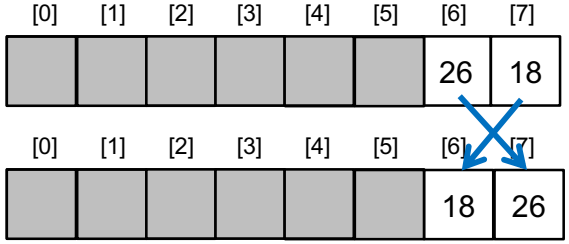


ソートの例

①—6

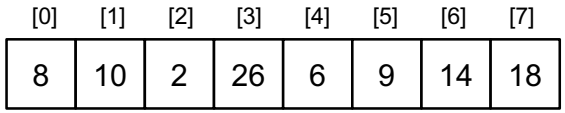


①—7

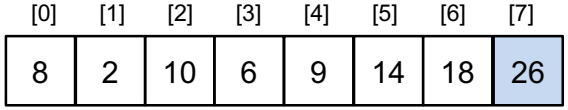


ソートの例

- 最初の状態



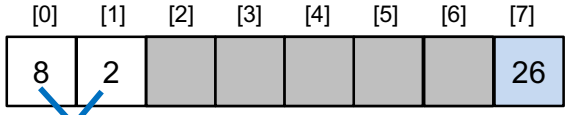
- ①が終わった状態



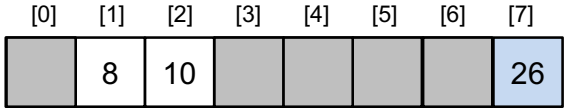
一番大きい数が必ず最後に来る

ソート例

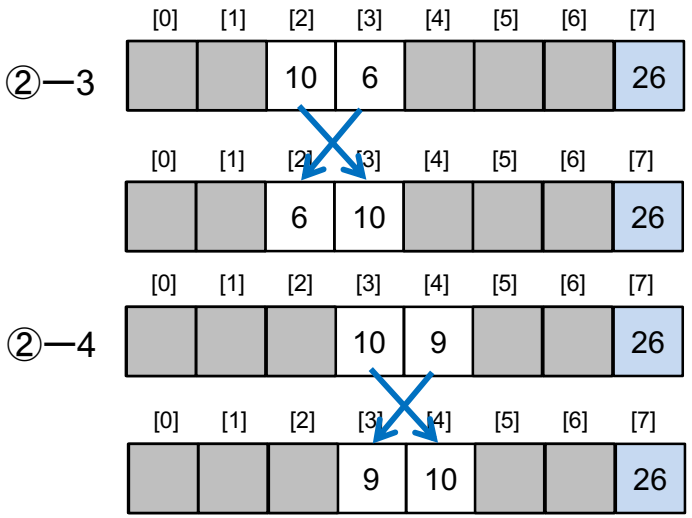
②—1



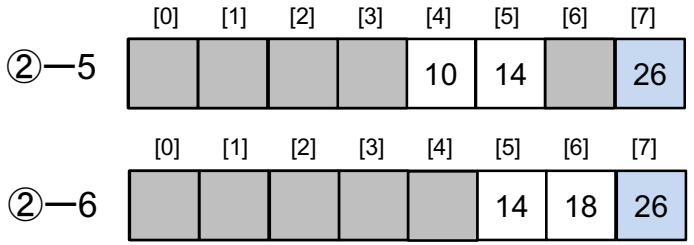
②—2



ソート例

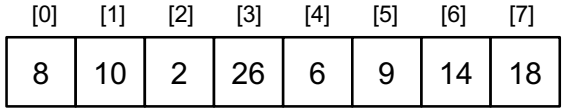


ソート例

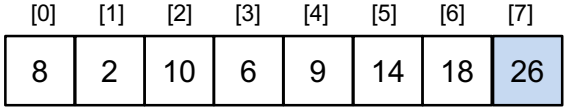


ソートの例

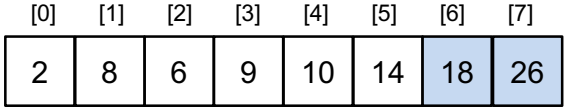
- 最初の状態



- ①が終わった状態



- ②が終わった状態



どれくらいの手数が必要か？その1

- ①が終わると，最後の要素に一番目に大きい数字が移動してきた
- ②が終わると，最後から一個手前の要素に二番目に大きい数字が移動してきた

- 要素数が8つのときは，①，②，…と同じようなことを8回くらいやればよさそう
  - 正確には7回（2つのときに1回やればよいので）

最悪のケース

初期状態	26	18	14	10	9	8	6	2
①終了時	18	14	10	9	8	6	2	26
②終了時	14	10	9	8	6	2	18	26
③終了時	10	9	8	6	2	14	18	26
④終了時	9	8	6	2	10	14	18	26
⑤終了時	8	6	2	9	10	14	18	26
⑥終了時	6	2	8	9	10	14	18	26
⑦終了時	2	6	8	9	10	14	18	26

どのくらいの手数が必要か？その2

- ①は7回の比較と入れ替えが必要だった
  - 対象となる要素数は8（元の要素数−1）
- ②は6回の比較と入れ替えが必要だった
  - 対象となる要素数は7（元の要素数−2）
- ①, ②, ③, ④, …はそれぞれ

  元の要素数 − 番号

  だけの回数処理が必要

どれくらいの手数が必要か?

- 要素数を n とすると
  - $n-1$回の**最大値を求める処理**が必要
    （今までの説明で①, ②, …と呼んだ処理）
  - m回目の最大値を求める処理は,
    ***n − m* 回の比較と入れ替え**が必要
- 全体で $\sum_{m=1}^{n-1} n-m$ 回の処理が必要

  だが, おおよそ n^2 に比例する回数の処理が必要となる
  - n回くらい必要な処理を, n回くらいするから

バブルソート

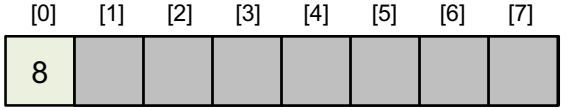
- 今まで見てきたソートの方法は
  バブルソートという
  - 最も単純だが（ほとんどの場合）効率が悪い
- 工夫次第で手数（計算量）を減らすことができる

その他のソート

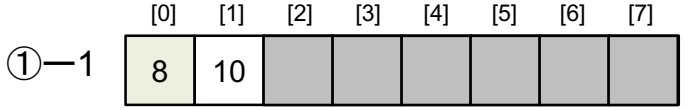
- 選択ソート
  - 最も大きい値を記憶し最後に移動させる(または最も小さい値を記憶し最後に移動させる)ということを繰り返す.
  - これも n^2 程度の手数が必要
- もう少しルールを緩めれば手数を減らせる
  - クイックソート
  - マージソート
  - ヒープソート

クイックソート

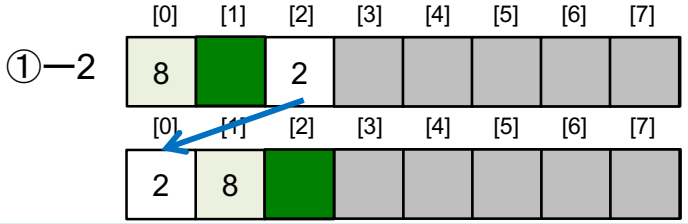
適当に基準となる数(ピボット)を決める



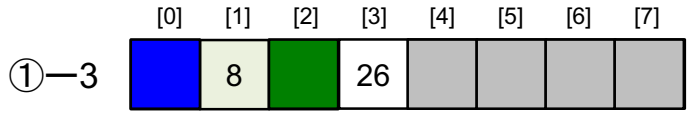
ピボットより大きいものは右側に持ってくる(この例の場合はそのまま)

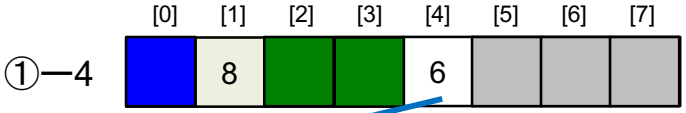


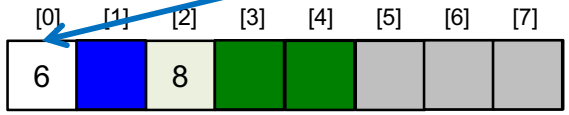
ピボットより小さいものは左側に持ってくる(2枚以上移動することになるので，ルール違反)

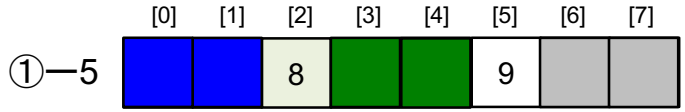


クイックソート

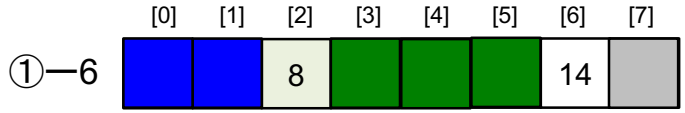


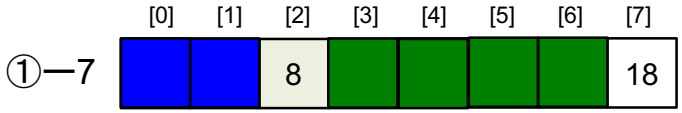


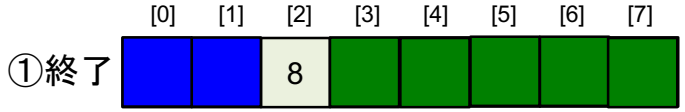


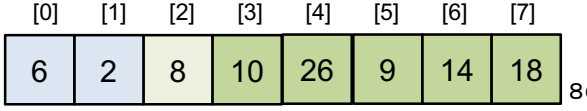


クイックソート



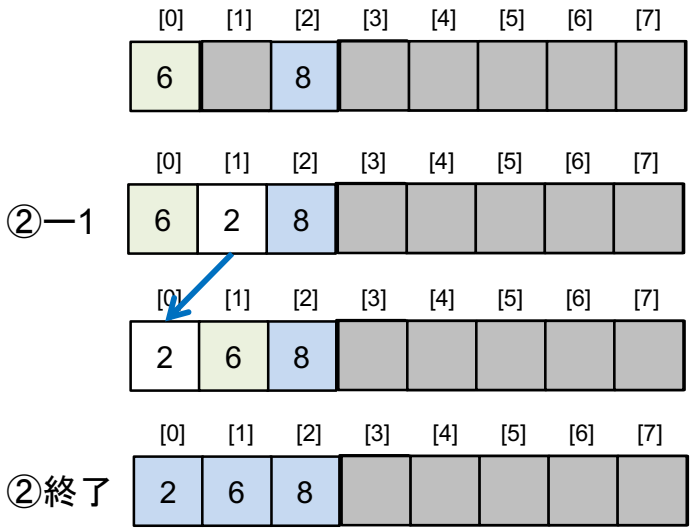




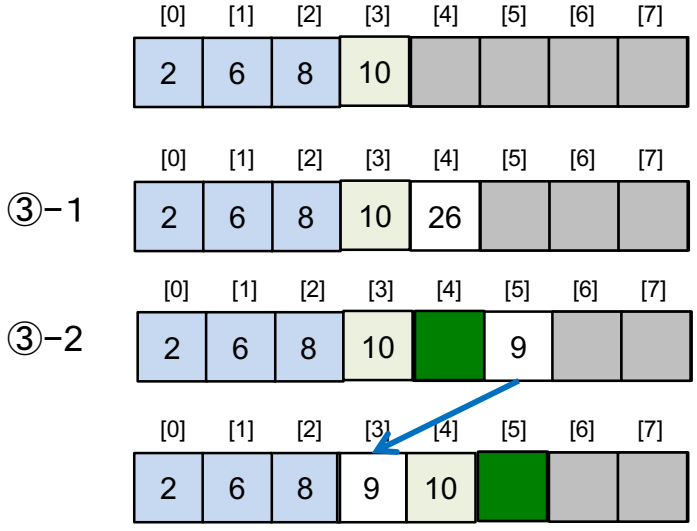


8の位置だけ確定

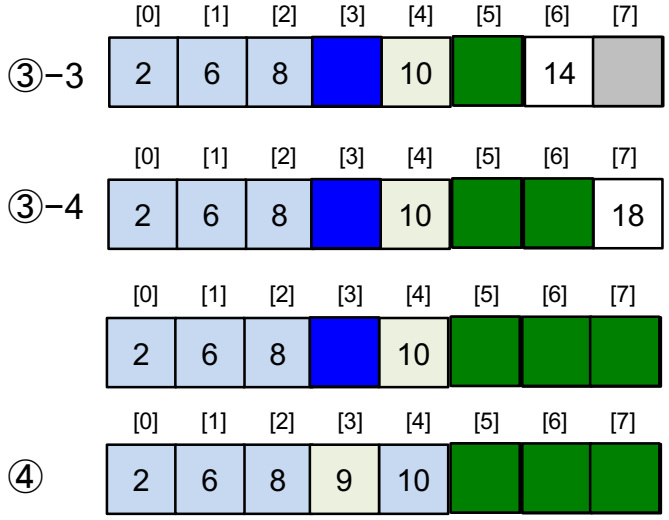
クイックソート



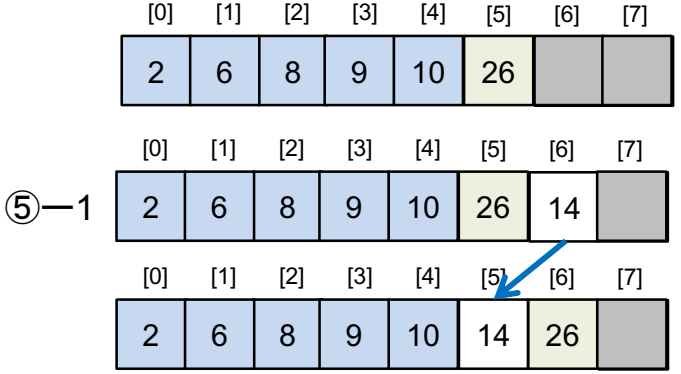
クイックソート



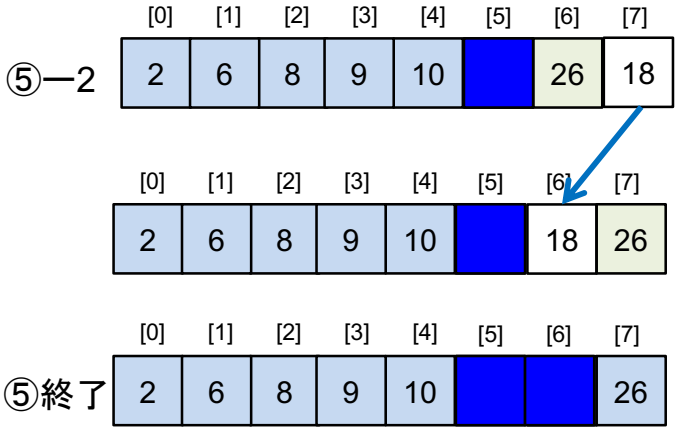
クイックソート



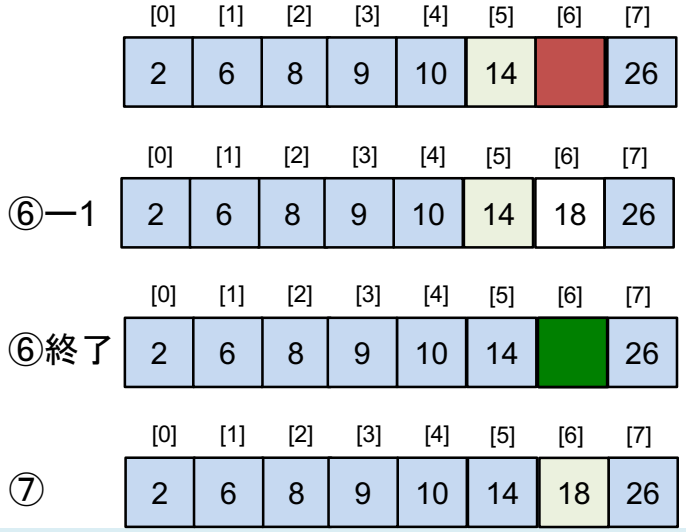
クイックソート



クイックソート



クイックソート



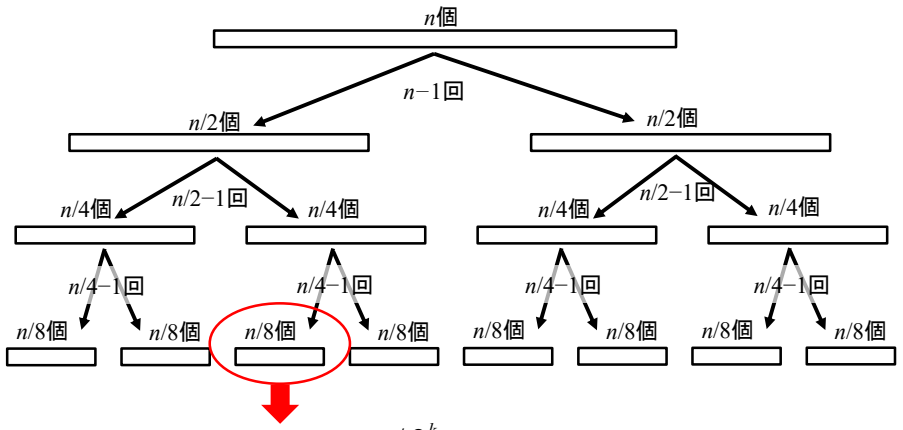
どれくらいの手数がかかるか?

- 仕分けをする処理(①, ②, ...)が*n*−1回ある
- 1回の仕分けにかかる処理は, 仕分け対象が*m*個あったら*m*−1 回かかる
- *n* 個の数値を仕分けすると(場合によるが) *n* / 2 個ずつの要素に分割されることが期待できる

- 全体 *n* 個
- 1段階目の仕分け(*n*−1回の比較)
  - *n*/2個, *n*/2個に分かれる
- 2段階目の仕分け(*n*/2−1回の比較を2回)
  - *n*/4個, *n*/4個, *n*/4個, *n*/4個に分かれる
- 3段階目の仕分け(*n*/4−1回の比較を4回)
  - *n*/8個, *n*/8個, *n*/8個, *n*/8個, *n*/8個, *n*/8個, *n*/8個, *n*/8個, に分かれる

...

- *k*段階目の仕分け($n/2^{k-1}-1$回の比較を2^{k-1}回)
  - $n/2^k$ 個の要素を持つ2^k通りに分かれる

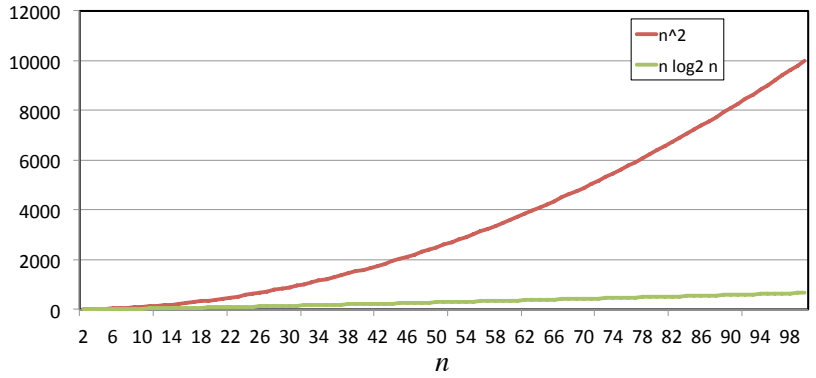


高さを k とすると，$n / 2^k$ 個．1個になるときのkを求めると...

$$n / 2^k = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$$

- 何段階目まで行くか
  - 要素数を n とすると，$\log_2 n$ 程度
- 1つの段階の処理はおおよそ何回の処理か
  - 要素数を n とするとおおよそ n 回
- つまり，全体としておおよそ $n \log_2 n$ 回程度かかることになる

n^2 と $n \log_2 n$ のどちらが大きいか?



nが大きくなるほど，差は大きくなる

配列

例題

- Aさん，Bさん，Cさんのテストの点数がそれぞれ80点，90点，70点のとき，3人の合計点，平均点をそれぞれ求めて表示するプログラムを作成せよ．

```
int a, b, c;
```



```
#include <stdio.h>

int main (void) {
  int a = 80;
  int b = 90;
  int c = 70;
  int sum = a + b + c;
  float avg = (float)sum / 3;

  printf("合計点: %d¥n", sum);
  printf("平均点: %.1f¥n", avg);

  return 0;
}
```


変数の限界

- 変数のメリット 
  - 変数を使ってメモリに数値を記憶できると，**計算の結果を使い回し**や，（scanf関数を使って）**数値を読み込み**計算に利用できる
- 変数の限界 
  - 1つ1つの変数は**無関係（名前が似てても…）**
  - 3人程度ならいいが，100人，1000人，….となったときに別々に変数を作らなければならない？？？

順番に並んだ数値を，順番を意識して使いたい！

配列

- **配列**（array）
  - 複数の数値をひとかたまり（一列）にして扱う
  - **一定の型**（整数，実数を問わない）の変数が**隙間なく順番に並んでいる**イメージ
  - 配列名と何番目の数値かで使う数値を指定する

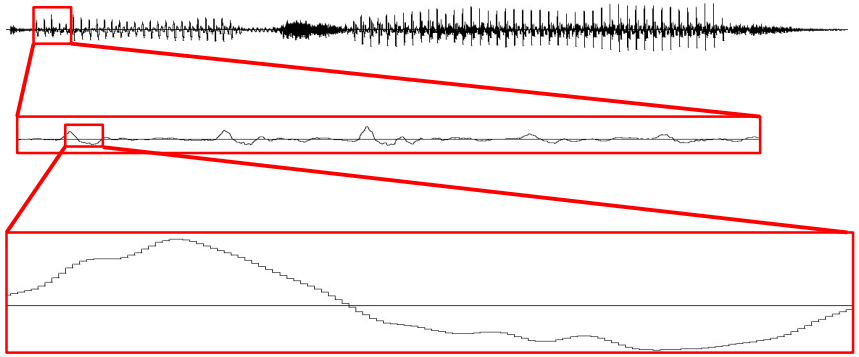
int型の配列 score

80	90	70

データの表現

- コンピュータの中ではあらゆる情報が数値の列

例)「こんにちは」という音声



一定間隔で音をサンプルしている

配列を使ったプログラム

```
#include <stdio.h>

int main (void) {
  int score[3];
  int sum, avg;

  score[0] = 80;
  score[1] = 90;
  score[2] = 70;

  sum = score[0] + score[1] + score[2];
  avg = (float)sum / 3;

  printf("合計点: %d¥n", sum);
  printf("平均点: %.1f¥n", avg);

  return 0;
}
```



```
#include <stdio.h>

int main (void) {
  int score[3];
  int sum, avg;

  score[0] = 80;
  score[1] = 90;
  score[2] = 70;

  sum = score[0] + score[1] + score[2];
  avg = (float)sum / 3;

  printf("合計点: %d¥n", sum);
  printf("平均点: %.1f¥n", avg);

  return 0;
}
```

配列の宣言文

```
int score[3];
```

型　配列名　要素数


```
#include <stdio.h>

int main (void) {
  int score[3];
  int sum, avg;

  score[0] = 80;
  score[1] = 90;
  score[2] = 70;

  sum = score[0] + score[1] + score[2];
  avg = (float)sum / 3;

  printf("合計点: %d¥n", sum);
  printf("平均点: %.1f¥n", avg);

  return 0;
}
```

要素の取り出し

```
score[0]
```

配列名

添字
（取り出したい要素の番号）

配列の要素数と添字

- `int score[3];` と宣言したときの要素数

int型の配列 score

???	???	???

要素数は3

- `int score[3];` と宣言したときの添字の範囲

int型の配列 score

0	1	2
???	???	???

添字（index）は0から始まるので，0~2となる

配列を使うメリット

- 3つの要素の配列を使うときに

```
sum = score[0] + score[1] + score[2];
```

などとしていても配列を活用できていない
（3つ別々の変数を使っているのと全く同じ）

- 配列は要素の取り出しのとき，
添字に変数を使うことができる
というのが利点

```
int score[3];
int i = 0;
score[i] = 80;
```



```
#include <stdio.h>

int main (void) {
  int score[3];
  int sum, i;
  float avg;

  score[0] = 80;
  score[1] = 90;
  score[2] = 70;

  sum = 0;
  for(i = 0; i < 3; i++) {
     sum += score[i];
  }
  avg = sum / 3.0;
  printf("合計点: %d¥n", sum);
  printf("平均点: %.1f¥n", avg);

  return 0;
}
```

array_avg.c

配列の初期化

- 変数宣言文での変数の初期化

```
int i = 0;
```

と同様に，配列も初期化が可能

- 配列の宣言文での初期化

```
int score[] = {80, 90, 70};
```

右辺に何個の要素があるかで要素数が自動的に
決まるので，ここは省略可能

多次元配列

- 今まで説明した配列は1次元配列
- 2次元配列，3次元配列，…が作れる
  - 2次元配列までは比較的よく使う
- 2次元配列
  - 例）サイズが2×3の2次元配列はこんな感じ

	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]

2次元配列の宣言と初期化

- 2次元配列の宣言

```
int a[2][3];
```

	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]

- 2次元配列の初期化

```
int a[][3]={{8,7,5}, {1,4,2}};
```

	0	1	2
0	8	7	5
1	1	4	2