コンピュータサイエンス第２
アルゴリズム：ソート（整列）

南出 靖彦

第２回

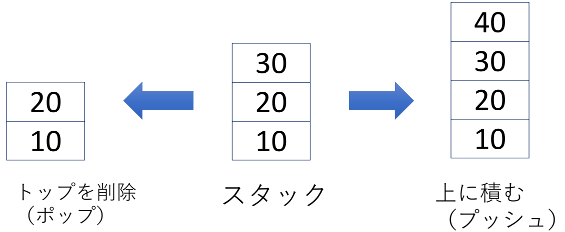
準備

- Documents フォルダの下の CS2 フォルダに移動
  - `cd Documents/CS2`
- 講義のウェブページから `day2.zip` をダウンロードする.
- Terminal で day2 を CS2 に移動
  - `mv ~/Downloads/day2 ./`
  - `cd day2`

再帰関数はどう実現されているか

メモリの一部をスタック として使う.

- スタック：データを積み上げたもの．伸びたり縮んだりする.

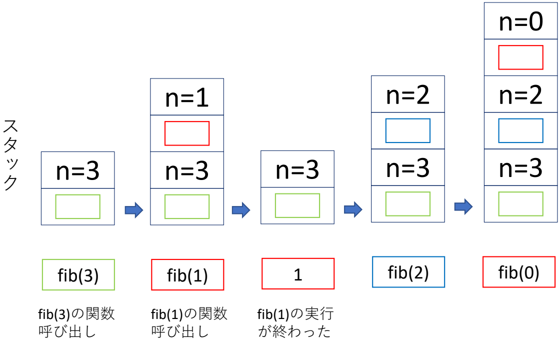


関数呼び出し：以下をスタックに積んで，関数本体を実行

- 関数実行後，プログラムのどこに戻るか
- 関数の実引数

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)

print( fib(3) )
```



アルゴリズム (algorithm)

アルゴリズムとは

- 計算手順
- 目標を達成する計算方法
- 良いアルゴリズム $\Leftrightarrow$ 効率の良いアルゴリズム
  - 計算時間が短い

整数に関するアルゴリズム

- 指数関数
- 最小公倍数
- フィボナッチ数
- n 番目の素数
- $\ldots$.

指数関数：二つのアルゴリズム

- 単純なアルゴリズム

$$n^k = n * n^{k-1}$$

$\Rightarrow$ k に比例した計算時間

- 工夫したアルゴリズム

$$\begin{aligned} n^{2k} &= (n^2)^k \\ n^{2k+1} &= n \times (n^2)^k \end{aligned}$$

$\Rightarrow$ $\log_2 k$ に比例した計算時間

注意：大きい数に対しては、本当は掛け算や割り算などの演算が定数時間ではできないので、上の計算時間は単純化しすぎ

- 単純なアルゴリズム

```
def exp1(n,k):
    r = 1
    for i in range(k):
        r = n * r
    return r
```

- 工夫したアルゴリズム

```
def exp2(n,k):
    if k == 0:
        return 1
    elif k % 2 == 0:
        return exp2(n * n, k // 2)
    else:
        return n * exp2(n * n, k // 2)
```

Python を対話的に使って試してみよう

```
$ python -i exp.py
>>> exp1(2,10)
1024
>>> exp2(2,10)
1024
>>>
```

第２引数が非常に大きいと実行時間の差がわかる

```
>>> exp1(2, 500000)
....
>>> exp2(2, 500000)
....
```

python の終わり方

```
>>> exit()
```

フィボナッチ数：計算時間

$fib(n)$: n 番目のフィボナッチ数

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n+2) &= fib(n+1) + fib(n) \quad (n \geq 0) \end{aligned}$$

$F(n)$: $fib(n)$ を実行した時に、関数 fib 何回呼ばれるか

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 1 \\ F(n+2) &= F(n+1) + F(n) + 1 \quad (n \geq 0) \end{aligned}$$

展開して計算すると

$$\begin{aligned} F(n+2) &= F(n+1) + F(n) + 1 = (F(n) + F(n-1) + 1) + F(n) + 1 \\ &> 2 \times F(n) \end{aligned}$$

計算時間：$F(n) \geq 2^{\frac{n}{2}}$ ($n \geq 3$ の時)

- n が少し大きくなると計算時間が長すぎる

フィボナッチ数：計算時間

```
$ python -i fib.py
>>> fib(10)
55
>>> fib(20)
6765
>>> fib(30)
832040
>>> fib(50)
```

止まらないので、Ctrl-C で止めてください。

注意：動的計画法の考え方を用いるアルゴリズムで，$fib(n)$ を n に比例する時間で計算できる

ソート（整列)

問題：

- 入力： 数列 $\langle a_1, a_2, \ldots, a_n \rangle$
- 出力: 入力を並べ替えた $\langle b_1, b_2, \ldots, b_n \rangle$
  - $b_1 \leq b_2 \leq \cdots \leq b_n$ が成り立つ

例：

- $\langle 8, 3, 1, 4, 6 \rangle \Longrightarrow \langle 1, 3, 4, 6, 8 \rangle$
- $\langle 8, 3, 1, 3, 6 \rangle \Longrightarrow \langle 1, 3, 3, 6, 8 \rangle$
  - 入力に同じ値が複数個あってもよい.

アルゴリズム (algorithm)

アルゴリズムとは

- 計算手順
- 目標を達成する計算方法
- 良いアルゴリズム $\Leftrightarrow$ 効率の良いアルゴリズム
  - 計算時間が短い

例えば，ソートには色々なアルゴリズムがある．数列の長さを n とする．

- バブルソート：計算時間が n^2 に比例
- マージソート：計算時間が $n \log n$ に比例
- $\cdots$

ソーティングゲーム

ウェブブラウザ上でソーティングゲームをやってみよう．

- http://www.e.gsic.titech.ac.jp/~kashima/SortGame/cs.html
  - まずは，「ランダム５枚」でやってみて，一般的な解法を考えよ。
  - 特に比較回数（画面左上に表示される）がなるべく少ない方法を考えよ。平均してどの程度の比較回数で解くことができるだろうか？
  - 効率の良い方法を考えたら「ランダム８枚」でやってみよう．

バブルソート (bubble sort)

以下を $k = n, (n-1), \ldots, 2$ に対して順に行う.

- $\langle a_1, a_2, \ldots, a_k \rangle$ に対して次のバブル手続きを行う（パスと呼ぶことにする）
  1. まず a_1 と a_2 の値を比較して，逆転していたら（つまり $a_1 > a_2$ だったら）この二要素を交換する
  2. 次に a_2 と a_3 の値を比較して逆転していたら交換する.
  3. このような隣接要素の比較・交換作業を「a_{k-1} と a_k」まで続ける
  4. k 個の中の最大値が a_k に浮かび上がってくる.

例：バブルソート

3	2	4	1	
				比較して交換
2	3	4	1	
	↓	↓		
2	3	4	1	
				パス 1 終わり，最大値が浮かぶ
2	3	1	4	
↓	↓			パス 2: ３番目の要素までを処理
2	3	1	4	
				パス 2 終わり
2	1	3	4	
1	2	3	4	

バブルソート：要素の交換

```
# 配列 a の i 番目と j 番目の要素を交換
def swap(a, i, j):
    tmp = a[i]
    a[i] = a[j]
    a[j] = tmp
    return a
```

バブルソート：バブル手続き

$\langle a_1, a_2, \ldots, a_k \rangle$ に対して次のバブル手続きを行う

1. まず a_1 と a_2 の値を比較して，逆転していたら（つまり $a_1 > a_2$ だったら）この二要素を交換する
2. 次に a_2 と a_3 の値を比較して逆転していたら交換する.
3. このような隣接要素の比較・交換作業を「a_{k-1} と a_k」まで続ける
4. i 個の中の最大値が a_k に浮かび上がってくる.

```
# 配列 a の先頭から k 個の要素にバブル手続きを行う
# a[0] 〜 a[k-1]
def bubble(a, k):
    for i in range(k-1):
        if a[i] > a[i+1]:
            swap(a, i, i+1)
    return a
```

注意: 列は a_1 から，配列は a[0] から

バブルソート：実行例

```
$ python -i bubble_sort.py
>>> swap([0,1,2],0,2)
[2, 1, 0]
>>> bubble([3,2,4,1],4)
[2, 3, 1, 4]
```

バブルソート：全体

以下を $k = n, (n-1), \ldots, 2$ に対して順に行う.

- $\langle a_1, a_2, \ldots, a_k \rangle$ に対して次のバブル手続きを行う

```
# 配列 a をバブルソートで整列
def bubble_sort(a):
    k = len(a)
    while k > 1:
        bubble(a, k)
        k = k - 1
    return a
```

実行例

```
>>> bubble_sort([3,2,4,1])
[1, 2, 3, 4]
```

バブルソート：二重ループ

手続き bubble の定義を，bubble_sort の中で展開する
⇒ 二重ループを持つプログラムになる

```
# 配列 a をバブルソートで整列
def bubble_sort(a):
    k = len(a)
    while k > 1:
        for i in range(k-1):
            if a[i] > a[i+1]:
                swap(a, i, i+1)
        k = k - 1
    return a
```

バブルソート：比較回数

$i = k$ の時

- a_1 と a_2 の比較
- a_2 と a_3 の比較
- $\dots$
- a_{k-1} と a_k の比較

$\Rightarrow k - 1$ 回比較を行う

$i = n, (n-1), \dots, 2$ に対して $\cdots$ なので

$$
\begin{aligned}
\text{合計の比較回数} &= (n-1) + (n-2) + \cdots + 1 \\
&= \frac{n(n-1)}{2}
\end{aligned}
$$

挿入ソート (insertion sort)

以下を $k = 2, 3, \ldots, n$ に対して順に行う.

1. すでに $\langle a_1, a_2, \ldots, a_{k-1} \rangle$ は整列している.
2. その列の適切な位置へ a_k を挿入することで，結果として左から k 個を整列させる.
3. 挿入結果を改めて $\langle a_1, a_2, \ldots, a_k \rangle$ と呼ぶ.

例

a_1	a_2	a_3	a_4	a_5	
7	$\boxed{3}$	2	4	6	$k = 2$ の時, 3 を挿入
3	7	$\boxed{2}$	4	6	$k = 3$
2	3	7	$\boxed{4}$	6	
2	3	4	7	$\boxed{6}$	
2	3	4	6	7	

```
# a[0] 〜 a[k-1] が整列済み, a[k] を挿入
def insert(a, k):
    tmp = a[k]
    i = k-1
    while i >=0 and a[i] > tmp:
        a[i+1] = a[i]
        i = i - 1
    a[i+1] = tmp
    return a
```

例: $\mathtt{k} = 4$

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	
2	3	5	7	$\boxed{4}$	
					$tmp = 4$
2	3	5	7	4	$i = 3$
2	3	5	7	7	$i = 2$
2	3	5	5	7	$i = 1$
					$a[1] \leq tmp$, tmp を $a[2]$ に代入
2	3	4	5	7	

挿入ソート：プログラム全体

以下を $k = 2, 3, \ldots, n$ に対して順に行う.

1. すでに $\langle a_1, a_2, \ldots, a_{k-1} \rangle$ は整列している.
2. その列の適切な位置へ a_k を挿入することで，結果として左から k 個を整列させる.
3. 挿入結果を改めて $\langle a_1, a_2, \ldots, a_k \rangle$ と呼ぶ.

```
def insertion_sort(a):
  for k in range(1, len(a)):
      insert(a, k)
  return a
```

挿入ソート：実行例

```
>>> insert([1,3,7,2],3)
[1, 2, 3, 7]
>>> insert([1,3,7,6,2],3)
[1, 3, 6, 7, 2]

>>> insertion_sort([3,2,5,1,4])
[1, 2, 3, 4, 5]
```

選択ソート (selection sort)

以下を $k = 1, 2, 3, \ldots, n-1$ に対して順に行う．

1. すでに $\langle a_1, a_2, \ldots, a_{k-1} \rangle$ は整列している．さらに，以下が成り立つ．

$$a_{k-1} \leq a_j \ (j \geq k)$$

2. $\langle a_k, a_{k+1}, \ldots, a_n \rangle$ の最小値のインデックス i を見つける
3. a_i と a_k を交換する
4. $\langle a_1, a_2, \ldots, a_k \rangle$ は整列している．

$$a_k \leq a_j \ (j \geq k+1)$$

選択ソート: プログラム概略

```
# a[k] 〜 a[n-1] の最小値のインデックスを返す
# ただし, n = len(a)
def find_min(a, k):
    min = a[k]
    index = k
    for i in range(k+1, len(a))
        # ここを埋める
    return index

# 配列 a を選択ソートで整列する
def selection_sort(a):
    for k in range(0, len(a)-1)
        # ここを埋める
    return a
```

選択ソート: 実行例

```
$ python -i selection_sort.py
>>> find_min([7,5,2,4,6,3],0)
2
>>> find_min([7,5,2,4,6,3],3)
5
>>> selection_sort([7,5,2,4,6,3])
[2, 3, 4, 5, 6, 7]
```