アルゴリズム論

2010年度

三つの問題を Python で解く[2]

行列の積

- 行列の積
  - 入力：$n \times n$ の実数値行列
    $\boldsymbol{A} = A[i, j]$ および $\boldsymbol{B} = B[i, j]$
  - 出力：$\boldsymbol{C} = C[i, j] = \boldsymbol{A} \cdot \boldsymbol{B}$

- 課題 p1：行列の積
  行列 A の列数と行列 B の行数が同じであるとき，その積 $A \cdot B$ を計算するプログラムを書け。また，その計算時間を行列のサイズの関数として表せ。

$n \times n$ 行列同士の積

```
def multiple(A, B):  # A と B は同じサイズの正方行列
    """ A と B の積を返す """
    size = len(A)     # len(A) はリスト A の長さ
    C = []            # C はリスト（初期値は空）と宣言
    for i in range(size):   # range(size) は [0,1,...,size-1]
        row = []      # row はリスト。C の i 行目を作る準備
        for j in range(size):  # j = 0, 1,..., size-1
            val = 0  # C の i 行 j 列目の計算準備
            for k in range(size):
                val += A[i][k] * B[k][j]
            row.append(val)   # リスト row の最後に val を追加
        C.append(row)         # リスト C の最後に row を追加
    return C
```

multipleAdv(A, B)

```
def multipleAdv(A, B):
    """ A と B の積を返す """
    rowA, colA = (len(A), len(A[0]))     # A の行数と列数
    rowB, colB = (len(B), len(B[0]))     # B の行数と列数
    if colA != rowB:
        return []          # 何らかのメッセージを出力するのもよい
    C = []                 # C はリスト（初期値は空）と宣言
    for i in range(rowA):            # rowA は積 C の行数
        row = []           # row はリスト。C の i 行目を作る準備
        for j in range(colB):        # colB は積 C の列数
            val = 0                 # C の i 行 j 列目の計算準備
            for k in range(colA):   # colA は rowB と同じ
                val += A[i][k] * B[k][j]
            row.append(val)   # リスト row の最後に val を追加
        C.append(row)         # リスト C の最後に row を追加
    return C
```

タプル in Python

◎ 要素をカンマで区切り，丸カッコで囲ったもの

◎ リストとよく似ているが，要素を書き換えることはできない

◎ 丸カッコは省略できる

☆ スワップ：x，y = (y，x)

［C言語なら　dummy = x; x = y; y = dummy;］

☆ 複数の値を戻すとき：return (x，y，z)

[return x，y，z でも可]

multipleAdv の計算時間

$A \in \mathbb{R}^{\ell\times m},\ B \in \mathbb{R}^{m\times n}$ とすると

［課題 p1］ $T(\ell, m, n) = a + \ell(b + n(c + md)))$
$= d\ell mn + c\ell n + b\ell + a$

$\mathbb{R}$ は実数の集合［他に, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{C}$］

$$\mathbb{R}^{\ell} = \overbrace{\mathbb{R}\times\mathbb{R}\times\cdots\times\mathbb{R}}^{\ell} = \{(x_1, x_2, \cdots, x_\ell) \mid x_k \in \mathbb{R}, 1 \le k \le \ell\}$$

$\mathbb{R}^{\ell\times m}$ は, ℓ 行 m 列の実数値行列の集合

$$\mathbb{R}^{\ell\times m} = \left\{ \begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_m \end{pmatrix} \middle|\ y_k \in \mathbb{R}^{\ell}, 1 \le k \le m \right\}$$

ソーティング

- ソーティング
  - 入力：n 個の実数列 $A = (a_1, a_2, \ldots, a_n)$
  - 出力：これらの数を非減少順に並べ替えた列

```
def bubbleSort(A):
    for i in range(len(A) - 1):
        for j in range(len(A) - 1, i, -1):
            if A[j - 1] > A[j]:
                A[j - 1], A[j] = (A[j], A[j - 1])
    return
```

最大と２番目

- 最大と２番目に大きい要素
  - 入力：n 個の正整数の集合 $A = \{a_1, a_2, ..., a_n\}$
  - 出力：最大の要素 a_{max} および２番目に大きい要素 a_{max2}

```
def bubble2(A):
    for i in [0, 1]:
        for j in range(len(A) - 1, i, -1):
            if A[j - 1] < A[j]:
                A[j - 1], A[j] = (A[j], A[j - 1])
    print "max =", A[0], "..  max2 =", A[1]
    return
```

課題 s1:バブルソートの計算時間

スワップ操作に要する時間を c とし，その他の演算時間は考えなくてよい。

(1) $T_{\mathrm{worst}}(n)$：与えられた数列が最悪の場合。どのような数列が最悪かも答えよ。

(2) $T_{\mathrm{average}}(n)$：数列がランダムであって，if 文が成立する確率が 1/2 である場合。

(3) bubble2 の最悪計算時間：$T_{\mathrm{bubble2}}(n)$

課題 s1:バブルソートの計算時間

```
def bubbleSort(A):    # n = len(A)
    for i in range(len(A) - 1):
        for j in range(len(A) - 1, i, -1):
            if A[j - 1] > A[j]:
                A[j - 1], A[j] = (A[j], A[j - 1])    # スワップ
    return
```

$$T_{\text{worst}}(n) = c\sum_{i=0}^{n-2}(n-1-i) = c\{(n-1)+(n-2)+\cdots+1\}$$

$$= c\sum_{k=1}^{n-1} k = \boxed{\frac{c}{2}n(n-1)}$$

$$T_{\text{average}}(n) = \frac{T_{\text{worst}}(n)}{2} = \boxed{\frac{c}{4}n(n-1)}$$

$$T_{\text{bubble2}}(n) = c\sum_{i=0}^{1}(n-1-i) = c\{(n-1)+(n-2)\} = \boxed{c(2n-3)}$$

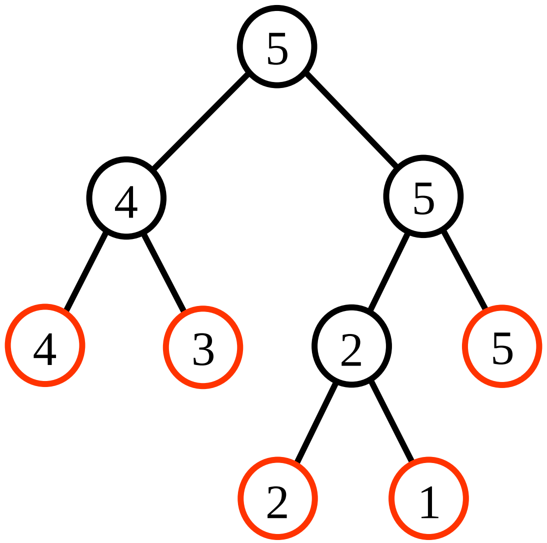
バブル法は勝ち残り戦

```
>>> A = [4,3,2,1,5]
>>> for j in range(len(A) - 1, 0, -1):
...     if A[j - 1] < A[j]:
...         A[j - 1], A[j] = (A[j], A[j - 1])
...     print A

[4, 3, 2, 5, 1]  # ← 1 対 5 は 5 の勝ち
[4, 3, 5, 2, 1]  # ← 2 対 5 は 5 の勝ち
[4, 5, 3, 2, 1]  # ← 3 対 5 は 5 の勝ち
[5, 4, 3, 2, 1]  # ← 4 対 5 は 5 の勝ち
```

優勝者をトーナメント方式で決めると ...

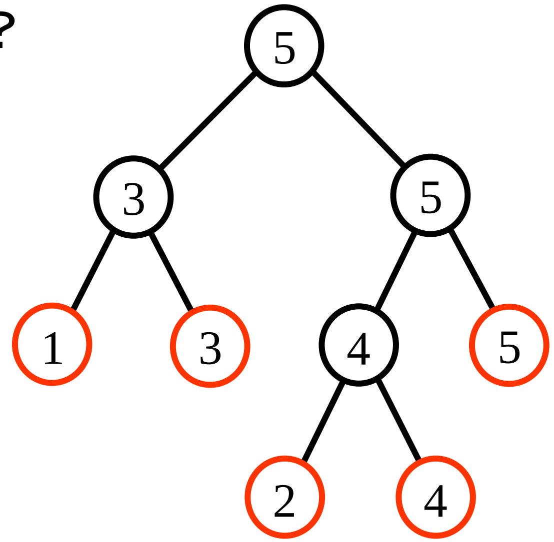
トーナメント方式



$n-1$ 回の試合（比較）で優勝者（最大）が決まる

２番目は，決勝戦の敗者？

２番目は，優勝者に
負けた者（値）の中に



優勝者の試合数は高々 $\lceil \log_2 n \rceil$

比較回数が高々（最悪でも）

$$(n-1)+(\lceil \log_2 n \rceil -1) = n + \lceil \log_2 n \rceil - 2$$

回のアルゴリズムが可能になる

bubble2 の比較回数は $(n-1)+(n-2)$

対数 logarithm

正の実数 $a \neq 1$ をとると, 任意の正の実数 x に対し
$x = a^p$ を満たす実数 p が唯一定まる。
この p を $p = \log_a x$ と書き, a を底とする x の対数という。

［常用対数/ブリッグスの対数］common logarithm
$a = 10$ であり, $\mathrm{Log}\, x$ と書くことがある

［自然対数/ネイピアの対数］natural logarithm
$a = e$ であり, $\ln x$ と書くことがある

［二進対数］binary logarithm
$a = 2$ であり, $\lg x$ と書くことがある

単に $\log x$ とあれば, 前後の文脈や扱われている分野から判断する

二進対数

$2^{\log x} = x$ であるから

$$2^{\log 2} = 2 \Rightarrow \log 2 = 1 \qquad 2^{\log 8} = 8 \Rightarrow \log 8 = 3$$

$$2^{\log 1024} = 1024 \Rightarrow \log 1024 = 10$$

$$2 < \log 5 < 3 \qquad 9 < \log 1000 < 10$$

$$\boxed{\log_a x = \frac{\log x}{\log a} \quad \text{for any base}}$$

Python では [モジュール math に定義されている]:

```
    log(x[, base]) -> the logarithm of x to the given
      base.  If the base not specified, returns the
      natural logarithm (base e) of x.
```

床関数と天井関数

床関数：実数 x に対し, x 以下の最大の整数　$\lfloor x \rfloor$ or $[x]$ or $\mathrm{floor}(x)$

天井関数：実数 x に対し, x 以上の最小の整数　$\lceil x \rceil$ or $\mathrm{ceil}(x)$

以下の性質において, x は任意の実数, k は任意の整数

$$\lceil x \rceil - 1 \le \lfloor x \rfloor \le x \le \lceil x \rceil \le \lfloor x \rfloor + 1 \qquad \lceil x \rceil = -\lfloor -x \rfloor,\ \ \lfloor x \rfloor = -\lceil -x \rceil$$

$$\left\lfloor \frac{k}{2} \right\rfloor + \left\lceil \frac{k}{2} \right\rceil = k \qquad [x \text{ を小数点以下四捨五入}] = \begin{cases} \lfloor x + 0.5 \rfloor & \text{if } x \ge 0 \\ \lceil x - 0.5 \rceil & \text{if } x \le 0 \end{cases}$$

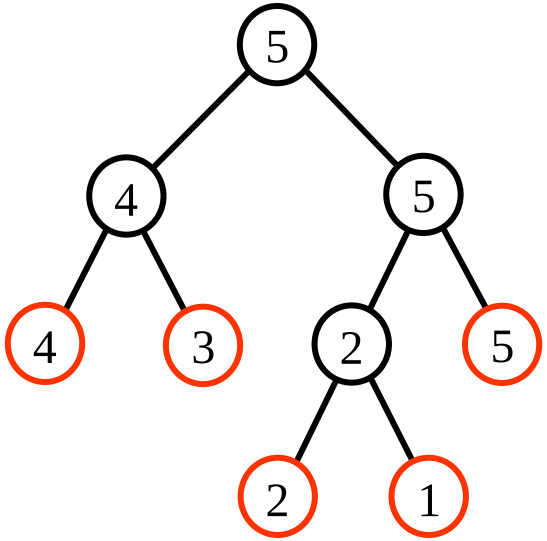
$$\lfloor k + x \rfloor = k + \lfloor x \rfloor \qquad k \le x \Leftrightarrow k \le \lfloor x \rfloor$$

$$\lceil k + x \rceil = k + \lceil x \rceil \qquad x \le k \Leftrightarrow \lceil x \rceil \le k$$

$$\lceil x \rceil - \lfloor x \rfloor = \begin{cases} 0 & \text{if } x \text{ は整数である} \\ 1 & \text{if } x \text{ は整数でない} \end{cases}$$

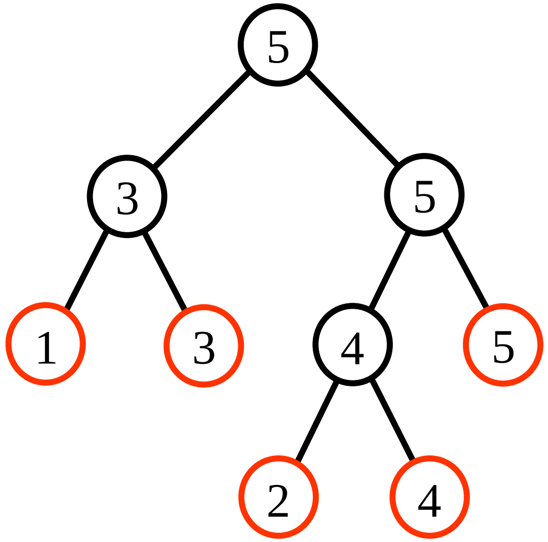
正の整数 k を n 進数で表すと桁数は $\lfloor \log_n k \rfloor + 1$ である

トーナメント方式



$n-1$ 回の試合（比較）で優勝者（最大）が決まる

２番目は，優勝者に
負けた者（値）の中に



優勝者の試合数は高々 $\lceil \log_2 n \rceil$

比較回数が高々（最悪でも）

$$(n-1)+(\lceil \log_2 n \rceil -1) = n + \lceil \log_2 n \rceil -2$$

回のアルゴリズムが可能になる

bubble2 の比較数は $(n-1)+(n-2)$

課題 s2：最大と２番目

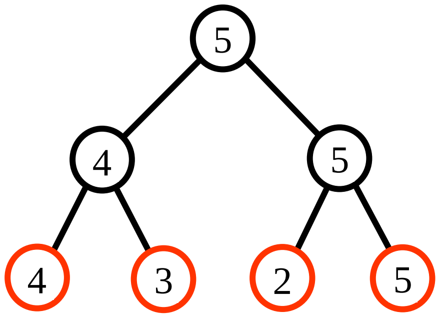
　与えられた n 個の正整数の集合から，最大と２番目の値を求める問題において，比較回数が高々 $n+\lceil \log_2 n \rceil$ の定数倍となるプログラムを書け。

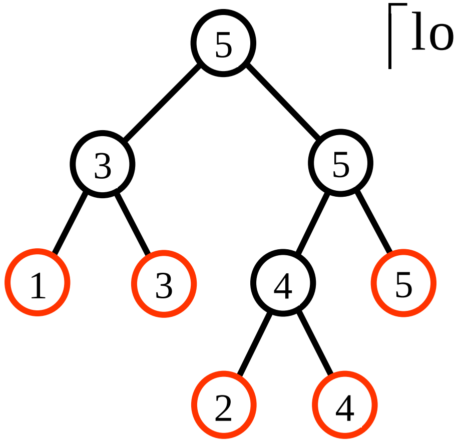
［ちょっと難しい？］

・n が２のべき乗でないとき，トーナメントの組み方はいろいろある

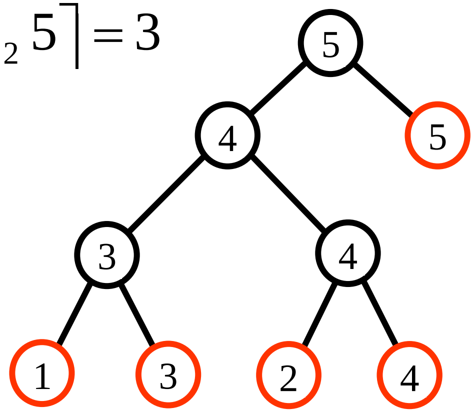
・２番目の値を計算しやすいデータ構造

トーナメントの組み方



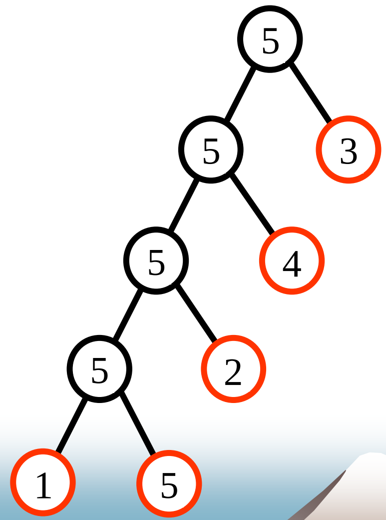


$$\lceil \log_2 5 \rceil = 3$$



```
def tourProto(A):
    if len(A) % 2 == 1:
        A.append(0)
    if len(A) == 2:
        return A
    halfL = tourProto(A[:len(A) / 2])
    halfR = tourProto(A[len(A) / 2:])
    return halfL + halfR
```

再帰：recursive

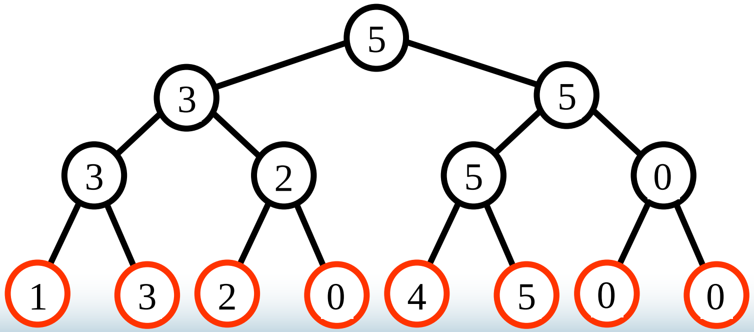


トーナメントの組み方

```
def tourProto(A):
    if len(A) % 2 == 1:
        A.append(0)
    if len(A) == 2:
        return A
    halfL = tourProto(A[:len(A) / 2])
    halfR = tourProto(A[len(A) / 2:])
    return halfL + halfR
```

２者を比較し
勝者を先頭へ

halfL の勝者と
halfR の勝者を
比較して，その
勝者を先頭へ



の部分を追加すれば
tounament が完成する

Max1and2(A)

```
def Max1and2(A):
    """ A の中の最大値と２番目の値を求める """
    A = tournament(A)
    print A                              # チェック用出力
    max1 = A[0]                          # A[0] は最大
    max2 = number2(A[:len(A) / 2], A[len(A) / 2])
                  # A[:len(A)/2] は最大の準決勝以下の戦歴
                  # A[len{A)/2] は決勝戦における敗者
    print "max1 =", max1, "  max2 =", max2
    return
```

Max1and2(A)

```
def tournament(A):
    """ A を２分割し, トーナメントを組む """
    if len(A) % 2 == 1:
       .....
       ..... tournament(...)


def number2(A, max2):
    """ ２番目を戻す：max2 は２番目候補。A には最大の戦歴 """
    if len(A) == 1:      # 最大の戦歴は調べつくした
        return max2
    else:
       ..... number2(...)
       .....
```

分割統治法 divide and conquer

(1) 与えられた問題をいくつかの小問題に分割し (divide)

(2) 各小問題の解を求め (conquer)

(3) 得られた小問題の解を用いて元の問題の解を得る

という操作を再帰的に繰り返して，アルゴリズムを作成する手法