アルゴリズム論

2017年7月10日

樋口文人

目次

- 再帰
  - 再帰と計算効率

再帰

再帰の構造

- 再帰プログラム
  - 部分的に自分自身を使って構成されているプログラム
  - 繰返し 自分自身を呼び出し続ける，for や while とは異なる繰返し構造
  - 繰返しを終了する条件が必要

```
recursiveFunc(n) {
    if (n == END) { return recursiveFunc()を使わない計算 }
    else {return recursiveFunc()を使った計算 }
}
```

再帰の例

- 階乗 n!

  if (n==0) { n!=1 }

  else { n!=n * (n-1)! }

- フィボナッチ数列

  if (n<2) { $F_1 = 1, F_0 = 0$ }

  else { $F_n = F_{n-1} + F_{n-2}$ }

- クイックソート
- 木構造の探索

クイックソート

- 問題を分割しながら繰返し同じ手続きを適用
  - 分割統治
  - 再帰（自分自身を呼び出す手続き）

```
program {
    function qs(array, left, right) {
        pivot = devide(array, array.length-1)
        l = qs(array, left, pivot-1)
        r = qs(array, pivot+1, right)
        return l + pivot + r
    }
    prepare(data)
    qs(data, 0, data.length-1)
}
```

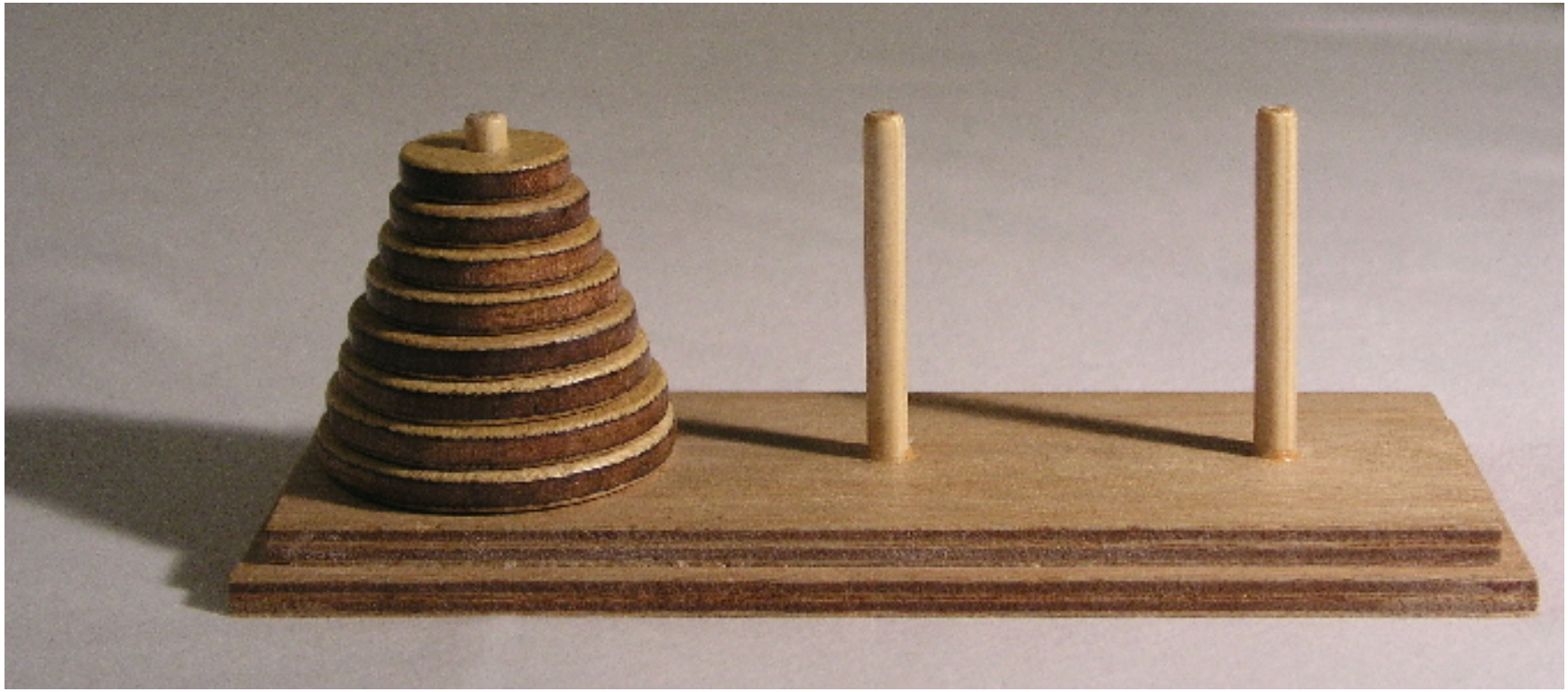
二分木の探索

1. [二分木の探索]
    1. 根の値と等しいとき: 探索終了
    2. 根の値より大きいとき:
        1. 右部分木があれば: [右部分木の探索]
        2. 右部分木がなければ: 探索終了
    3. 根の値より小さいとき:
        1. 左部分木があれば: [左部分木の探索]
        2. 左部分木がなければ: 探索終了

再帰的アルゴリズムの特徴

- 小規模な同じ問題の解を使って問題を解く
  - （より小規模な問題について）自分自身を呼び出す
  - 最も簡単な場合が存在
    - 再帰的な処理ではない解がある
  - 繰返し構造が表面に現れない

ハノイの塔



https://upload.wikimedia.org/wikipedia/commons/0/07/Tower_of_Hanoi.jpeg

Hanoi(n, a, b)

n 枚の山を a から b へ移動

```
function Hanoi(n, a, b) {
    Hanoi(n-1, a, c)
    Hanoi(1, a, b)
    Hanoi(n-1, c, b)
}
```

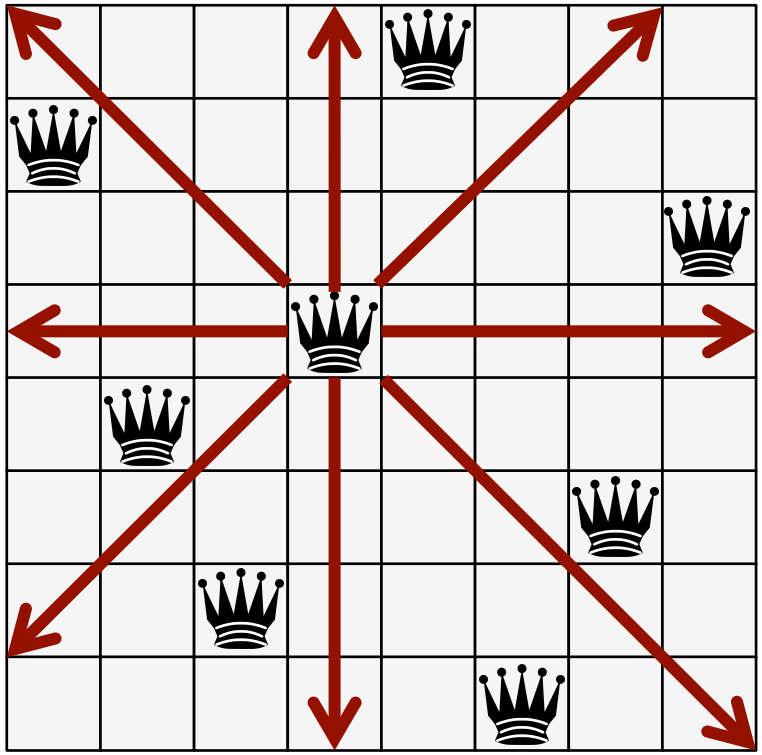
トラックバック

- プログラムで解く問題の中には試行錯誤をしながら力づくで解かなければならない問題がある
- 問題に係るパラメータの組み合わせを変えながら，１つの組み合わせがダメだったら1つ戻って別の組み合わせを試す
- 深さ優先探索と類似の方法

8-Queen

- 8x8 の盤面に8個のクイーンをお互いの効き筋（縦横斜め）に来ないように配置する
  - queen[8], board[8][8] の配列を使う
  - queen[i] = j とは，盤上の (i列, j行) にクイーンがある
  - board[i][j]≠ 0 とは盤上の位置 (i, j) には駒を置けない（他の駒の効き筋当たっている）
- N-Queen: NxNへの拡張問題

8-Queen

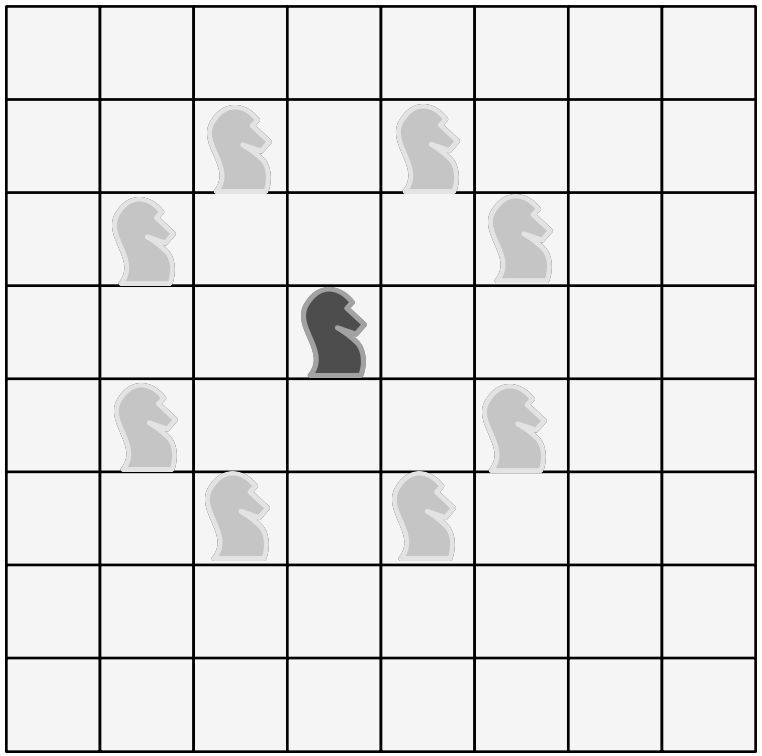


8-Queen の解法

- 繰返しを使う
  - ８重のループ
  - １２通りの独立解，９２通りの変形
- N-Queen
  - 繰返しは使えない
    - for や while では入れ子の深さを可変にはできない
  - バックトラックを使って再帰的に解く

ナイト巡回問題

- ナイトで8x8の全てのマスを1巡して元に戻る経路を見つける



再帰

- 問題を定式化するのに都合がよい
  - 簡潔に記述できる
    - 数学的に無限を扱える
    - プログラミング上は注意が必要
- 他の定式化が可能な場合がある
  - 繰返し
    - n! : f = 1; for (i = 0; i <n; i++) { f = f*(i+1) }; return f;
  - 直接計算
    - Fn : g = (1+√5)/2; return $(g^n-(-g)^{-n})/\sqrt{5}$

再帰のステップ

関数の呼び出し

$$n! = n \cdot (n-1)!$$
$$(n-1)! = (n-1) \cdot (n-2)!$$
$$(n-2)! = (n-2) \cdot (n-3)!$$
$$\vdots \quad \vdots$$
$$2! = 2 \cdot 1!$$
$$1! = 1 \cdot 0!$$
$$0! = 1$$

返り値を戻す

再帰ではない処理

関数の呼び出しとは

- プログラムの途中で関数を呼び出すと…
  - そのときのプログラムカウンタ，レジスタメモリの値，その他変数の値を退避（スタックに積む）
  - 関数の処理が終わると退避した情報を使って元の状態に戻した上で，プログラムの処理を継続
- 関数の処理の中で関数を呼び出すときも同様
  - 再帰の場合にも，変数その他の情報をスタックに積む
    - 同じ名前の変数でも，呼び出される度に異なる値を取るので（固有のメモリスペースを取る）別の変数と見做す

再帰の効率

- プログラムの記述
  - 単純で明快
  - 読みやすい
- 計算速度／メモリ消費
  - 通常，他の方法の方が効率的

フィボナッチ数列を再帰で計算

$$
\begin{aligned}
F_{10} &= F_9 + F_8 \\
&= (F_8 + F_7) + (F_7 + F_6) \\
&= \{(F_7 + F_6) + (F_6 + F_5)\} + \{(F_6 + F_5) + (F_5 + F_4)\} \\
&= [\{(F_6 + F_5) + (F_5 + F_4)\} + \{(F_5 + F_4) + (F_4 + F_3)\}] + [\{(F_5 \\
&\quad + F_4) + (F_4 + F_3)\} + \{(F_4 + F_3) + (F_3 + F_2)\}] \\
&= [\{((F_5 + F_4) + (F_4 + F_3)) + ((F_4 + F_3) + (F_3 + F_2))\} + \{((F_4 \\
&\quad + F_3) + (F_3 + F_2)) + ((F_3 + F_2) + (F_2 + F_1))\}] + [\{((F_4 + \\
&\quad F_3) + (F_3 + F_2)) + ((F_3 + F_2) + (F_2 + F_1))\} + \{((F_3 + F_2) + \\
&\quad (F_2 + F_1)) + ((F_2 + F_1) + (F_1 + F_0))\}]
\end{aligned}
$$

フィボナッチ数列（単純な再帰）

Python プログラム

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

print fibo(99)
```

実行結果

- 20分たっても答えがでない

対策

- メモ化
  - 既に計算した値を記憶しておく
  - 再帰が深くなることを防止する
- 末尾再帰
  - 処理の途中結果を再帰した関数のそれぞれが持つのではなく，引数として途中結果も再帰呼び尺をした関数に送る

フィボナッチ数列 (メモ化)

Python プログラム

```
rslt = [1, 1]

def fibo(n):
    if n < 0:
        return -1
    elif n < len(rslt):
        return rslt[n]
    else:
        rslt.append( fibo(n-1) + fibo(n-2) )
        return rslt[n]

print fibo(99)
print fibo(999)
```

実行結果

```
354224848179261915075
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037047613779516684922887 5
```

フィボナッチ数列 (末尾再帰)

Python プログラム

```
def fibo(n, a=1, b=1):
    if n < 0:
        return -1
    elif n == 0:
        return 1
    elif n == 1:
        return a
    else:
        return fibo(n-1, a+b, a)

print fibo(99)
print fibo(999)
```

実行結果

```
354224848179261915075
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037047613779516684922887 5
```

フィボナッチ数列 (末尾再帰)

コンパクトバージョン

Python プログラム

```
def fibo(n, p=1, val=1):
    return fibo(n-1, val, val + p) if n > 1 else val
print fibo(99)
print fibo(999)
```

実行結果

```
354224848179261915075
43466557686937456435688527675040625802564660517371780402481729
08953655417949051890403879840079255169295922593080322634775209
96896232398733224711616429964409065331879382989696499285160037
044761377951668492288 75
```

非再帰バージョン

フィボナッチ数列 (繰返し)

Python プログラム

```
def fibo(n):
    val, prev = 1, 1
    for _ in xrange(n-1):
        val, prev = val + prev, val
    else:
        return val
print fibo(99)
print fibo(999)
```

実行結果

```
354224848179261915075
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

スタックの限界

再帰

print fibo(1999)

```
 File "fibonacci.py", line 22, in fibo
    rslt.append( fibo(n-1) + fibo(n-2) )
RuntimeError: maximum recursion depth exceeded
```

繰返し

print fibo(1999)

```
42246963333923048787067256023414827825798528402506810980102801373143085843701307072241235996391415110884460875389096036076401947116435960292719833125987373262535558026069915859152294924539049987222567953169828744824729922639018337167780606070116154978867198798583114688708762645973690867228840236544222952433479644801395153495629720876526560695298064998419774487201756128026654045541717178819303240252043120825168171 25
```

再帰のステップ

関数の呼び出し

fibo(n) = fibo(n-1) + fibo(n-2)

fibo(n-1) = fibo(n-2) + fibo(n-3)

fibo(n-2) = fibo(n-3) + fibo(n-4)

. .

fibo(2) = fibo(1) + fibo(0)

fibo(1) = 1

fibo(0) = 1

再帰ではない処理

返り値を戻す

末尾再帰のステップ

関数の呼び出し

fibo(n, p=1, val=1) = fibo(n-1, val, val+p)

fibo(n, 1, 1) = fibo(n-1, 1, 2)

fibo(n-1, 1, 2) = fibo(n-2, 2, 3)

. .

もし n=10 なら

fibo(2, 21, 34) = fibo(1, 34, 55)

fibo(1, 34, 55) = fibo(0, 55, 89)

fibo(0, 55, 89) = 89

再帰ではない処理

返り値=再帰の結果

途中結果も引数として送る

おかしな関数

- アッカーマン関数

  Ack(m, n) =

  if m = 0 : n + 1

  if n = 0 : Ack(m-1, n)

  otherwise: Ack(m-1, Ack(m, n-1))

- 竹内関数（たらいまわし関数）

  Tarai(x, y, z) =

  if x ≤ y : y

  otherwise: Tarai(Tarai(x-1, y, z), Tarai(y-1, z, x), Tarai(z-1, x, y))

- マッカーシー版たらいまわし関数

  – Tak(x, y, z) =

    - if x ≤ y : z
    - otherwise: Tak(Tak(x-1, y, z), Tak(y-1, z, x), Tak(z-1, x, y))

再帰と計算効率

再帰による階乗計算

```
int factorial(int n) {
    if (n == 0) { return 1 } else { return n*factorial(n-1) }
}
factorial(10)
```

- n 回の再帰（関数呼び出し）
- スタックに n 回分の情報が積まれ，戻り時にn 回の乗算
- メモリの使用量：O(n)
- 演算ステップ数：O(n)

繰返しによる階乗計算

```
int factorial(int n) {
    if (n == 0) { return 1 } else {
        int f = 1
        for (int i = 1; i <= n; i++) { f = f*i }
        return f
    }
}
```

- n, 1, f, i をメモリの保持，n 回の繰り返しで乗算と代入を実行
- メモリの使用量：O(1)
- 演算ステップ数：O(n)

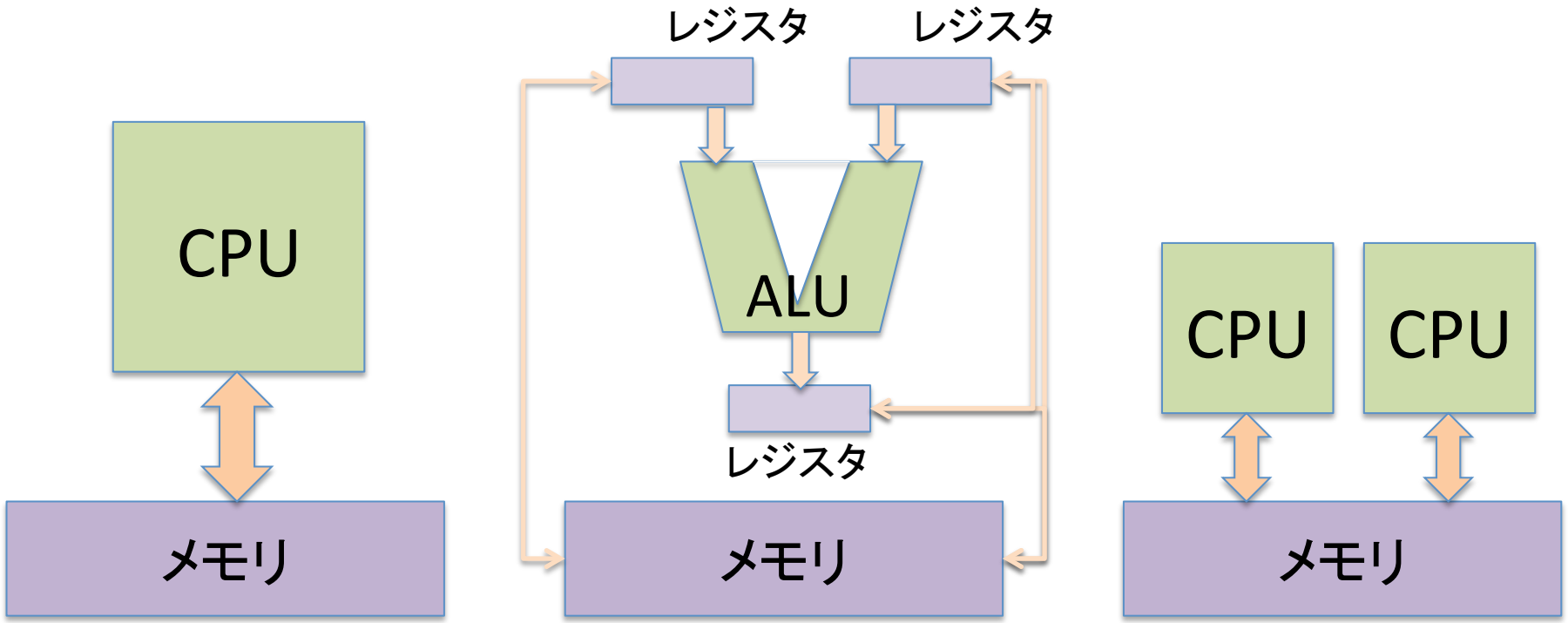
計算量

- 演算処理に必要なリソースの量
  - 時間：演算に必要なステップ数
  - 空間：使用するメモリ量
- 時間計算量と空間計算量の間にはトレード・オフの関係が成り立つことが多い
  - つまり，メモリを多く使うとステップ数が減る，
  - または，メモリを少なくしようとするとステップ数が増える

トレード・オフの例

- 配列と線形リスト
  - 隣接行列と隣接リスト
- バケットソート
  - 比較をしない
  - データの取り得る値の数だけ記憶場所が必要

メモリと演算装置



やってみよう:クイズ$_1$

1. Tarai(3, 2, 1) を計算して下さい

   Tarai(x, y, z) =

   if x ≤ y : y

   otherwise: Tarai(Tarai(x-1, y, z), Tarai(y-1, z, x), Tarai(z-1, x, y))

2. １からnまでの和を再帰を使って求めるプログラムを作って下さい
3. 再帰を使って最大公約数を求める

やってみよう:クイズ$_2$

- フローチャートを描いてください
  - １からNまでの整数の和を求める手順
  - 上記の手順をQ回繰り返す手順
  - 処理の最初から最後までに掛る時間を計測する手順を追加

やってみよう：クイズ$_3$

- ２つの３２ビット浮動小数点数の大小を比較する手順をフローチャートに書け
  - 符号：１ビット
  - 指数：８ビット
  - 仮数：２３ビット

やってみよう:クイズ$_4$

- 「1からNまでの和」を計算していた部分を「配列に格納したN個のデータの和」に修正したプログラムのフローチャートを描け
  - 配列要素に初期値を乱数で設定する
  - 和の計算はQ回繰り返す
  - Q回の繰り返し処理に掛かる時間を計測する

やってみよう:クイズ$_5$

1. 探索対象を見つけるまでの比較回数の期待値を確認してください
   1. ランダムな探索
   2. 線形探索

   データが整列しているかどうかの影響は？
2. １からNまでの数をシャッフルして探索する場合と，０から１までのN個の乱数データを探索する場合で比較回数の平均値に差があるのは何故か

やってみよう:クイズ$_6$

- ビット配列で集合を表すときどのような操作・演算により下記を実現できるか？
- 集合A, B について
  - A = B
  - A ⊂ B
- 要素 x について
  - x を A に加える
  - x を A から取り除く
  - x が A の中にあるかどうか探す

やってみよう：クイズ$_7$

- 今，１６ビットのビット配列である集合が表現されている．何個の要素を含んでいるか求める手順を考えよ．
  - ヒント
    - ４ビットの配列　0110 は第２要素と第３要素を含むことを表現しています
    - １ビットのビット配列で表された集合が４個並んでいると考えると，0110 は要素０個を含む集合が２つ，要素１個を含む集合が２つあることになります

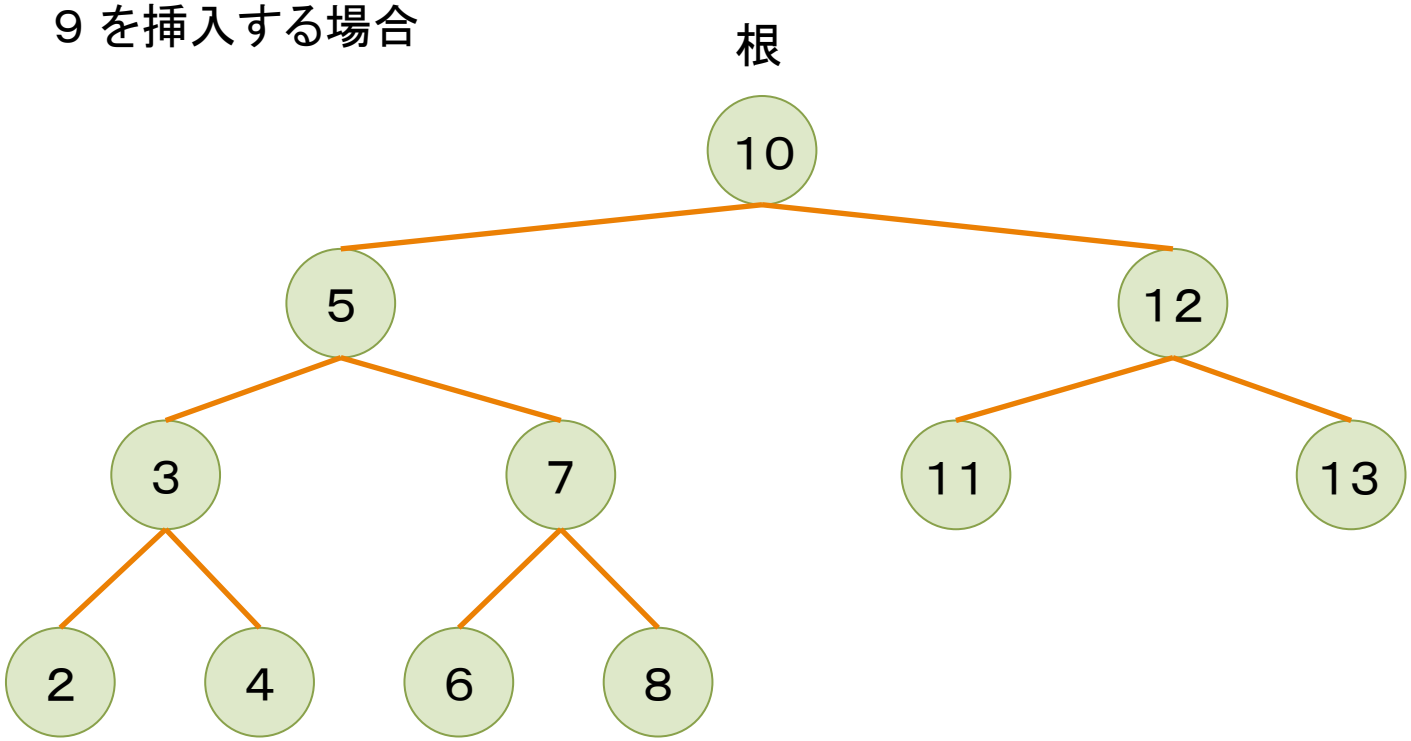
やってみよう:クイズ8

- cell の挿入や削除を行う関数またはメソッドはどのようにプログラムできるでしょうか？
- また、そのプログラムのコードでは，最初のデータの前や最後のデータの後にcellを挿入したり，これらのデータを削除する場合，特別な配慮が必要か？

やってみよう：クイズ9

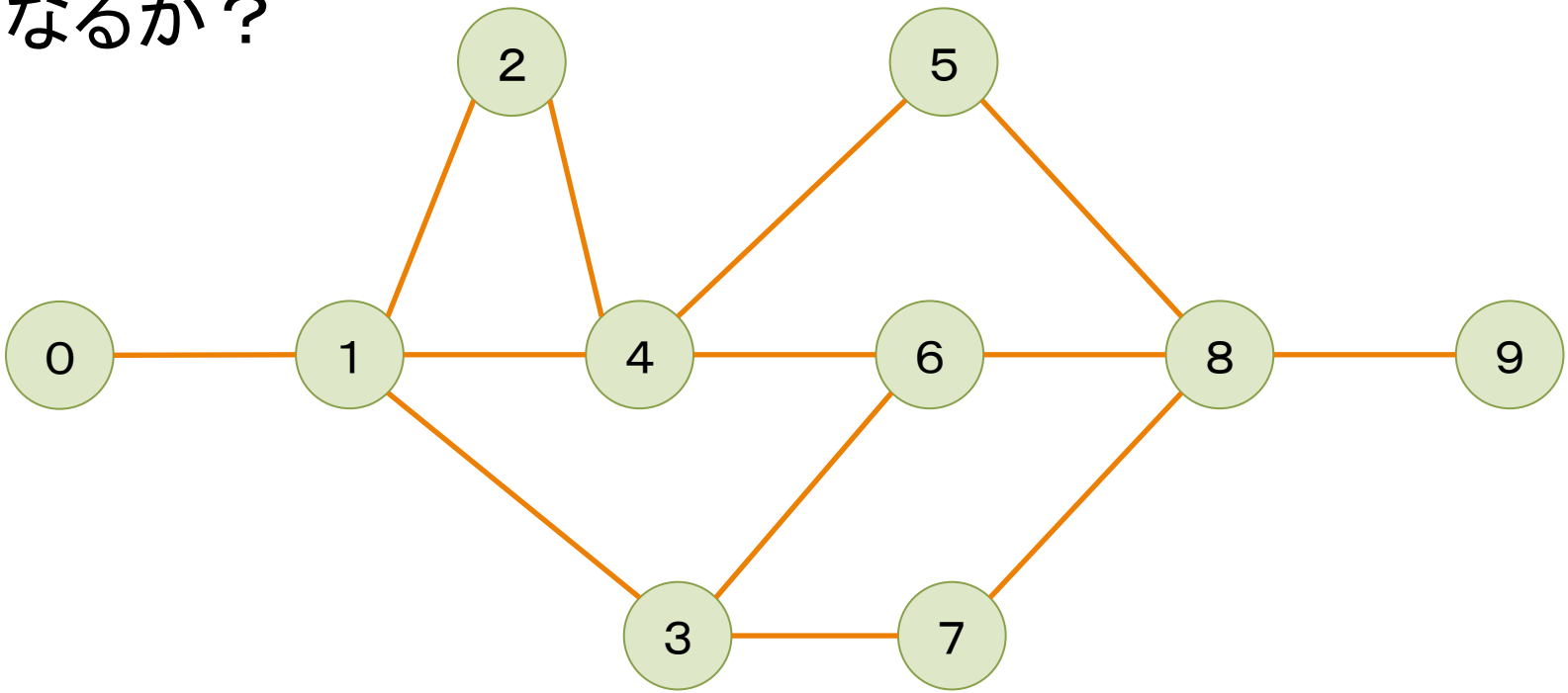
このAVL木に次の操作を行うとそれぞれ結果は？

- 1 を挿入する場合
- 9 を挿入する場合

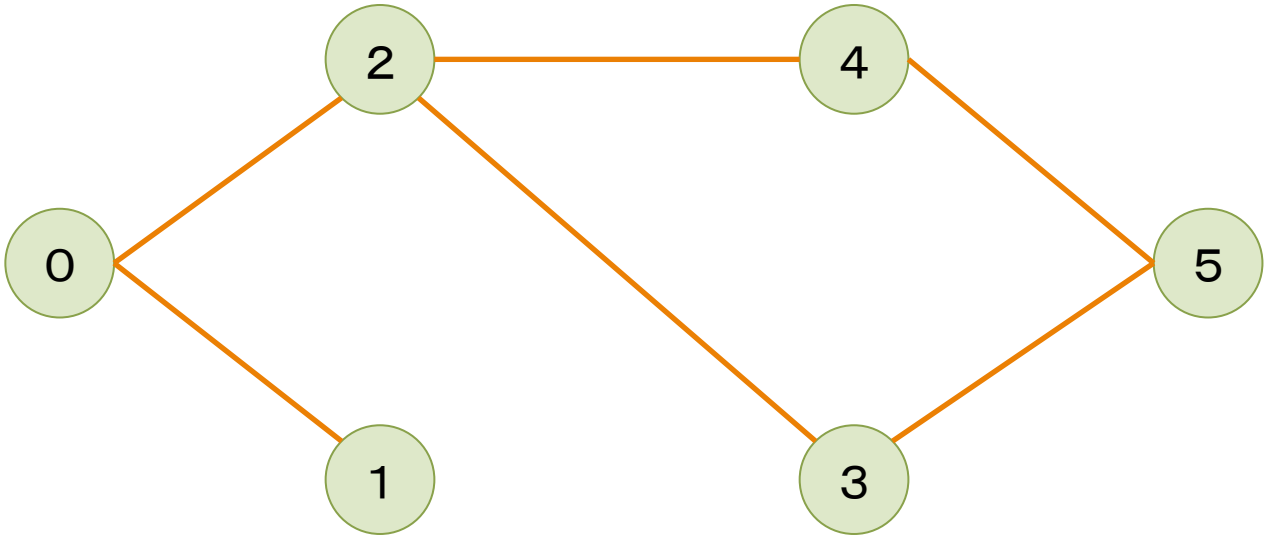


やってみよう:クイズ$_{10}$

- スタック(キュー)の初期状態として唯一0が入っているとして探索で訪問する節の順番はそれぞれどうなるか？

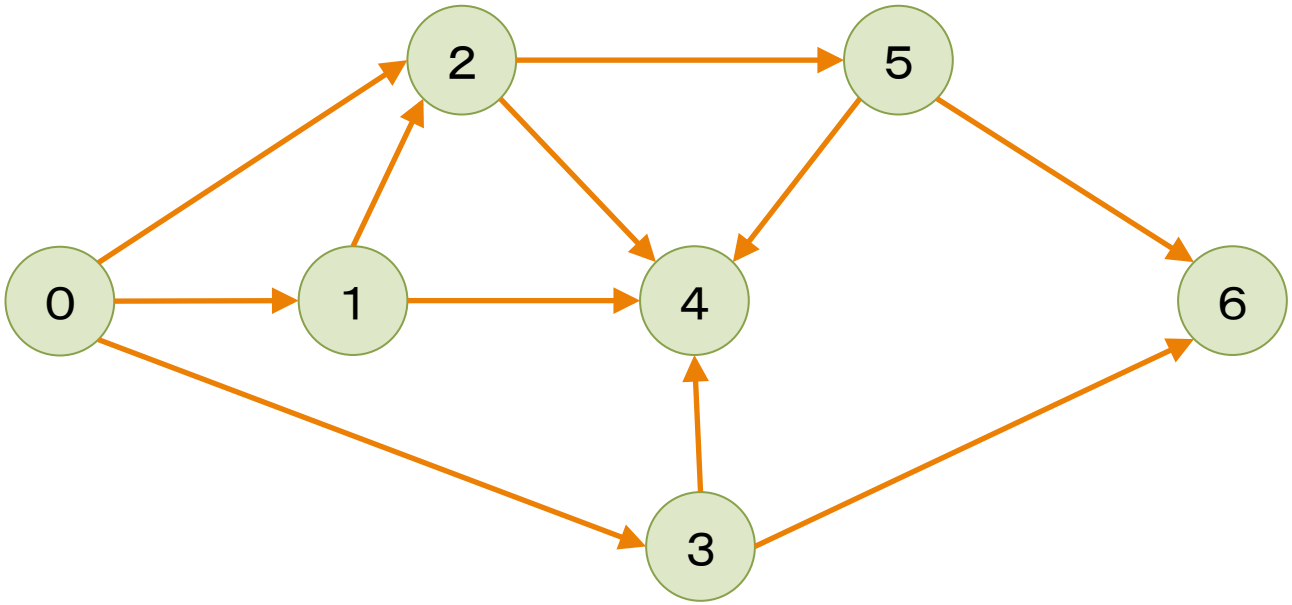


やってみよう:クイズ$_{11}$



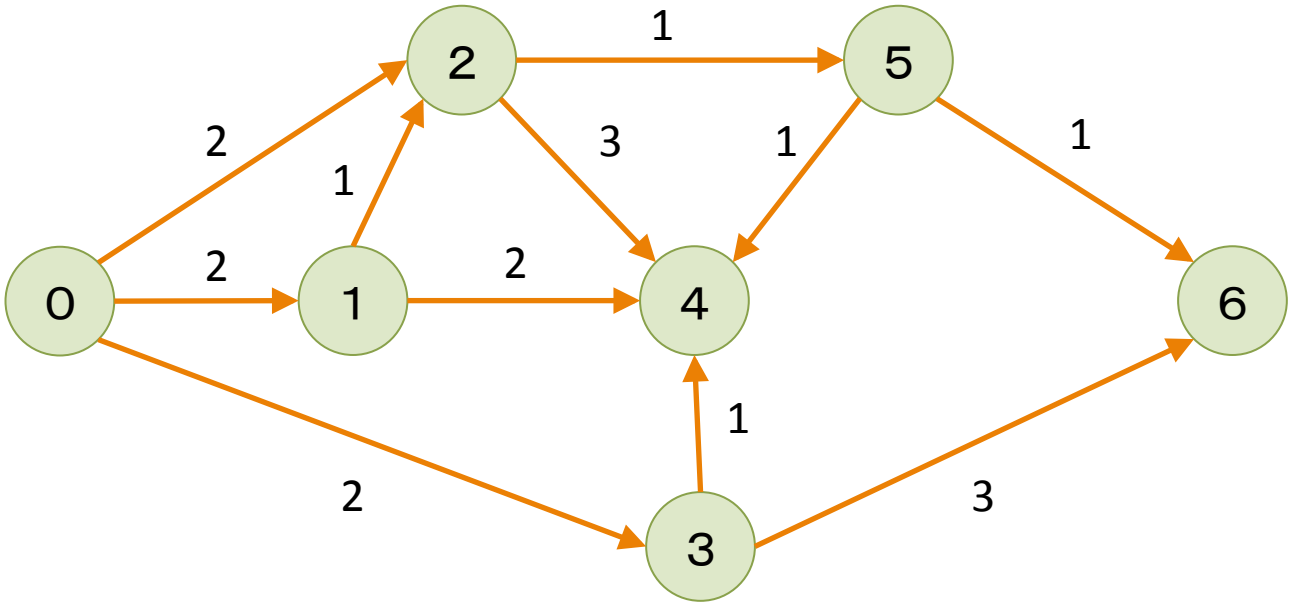
やってみよう：クイズ$_{12}$

トポロジカルソート



やってみよう:クイズ13

節点0からの距離



連絡先

樋口文人

wenren@meiji.ac.jp