アルゴリズム・データ構造 I 第3回
関数・再帰呼び出しと配列のしくみ

名城大学理工学部情報工学科
山本修身

この講義について

講義名： アルゴリズム・データ構造 I
担当者： 山本修身 (osami@meijo-u.ac.jp)
授業日：月曜日３時限（Bクラス），金曜日１時限（Aクラス）
教室： タワー7階 T-705
関連する科目： **プログラミング演習１，プログラミング演習２**
仮定する知識： プログラミングの基礎，プログラミングの経験
ホームページ： **http://osami.s280.xrea.com/Algo_Data2014/**
成績の評価： レポート（30%）と期末試験（70%）

Algorithm Data Structure

前回までに解説したプログラムのしくみ

- 式： データやデータどうしを演算子でつなぐ
  `2 * 3, 4 / 5, “abc”, ‘名城大学’, (444).toString(2)` など
- 変数：変数へデータを結びつけることができる
  `x = “名城大学”`
  x → 名城大学
- 制御構造（while, if, for）：繰返しや分岐を記述する．これにより複雑な動きを指定することができる．

```
s = ''
for (var i = 0; i < 10; i++)
   s += 'アルゴリズム'
puts(s)
```


もっと複雑なデータを扱おう

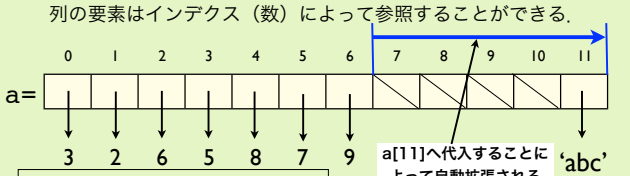
- 複雑なデータを扱うための仕組みとして配列がある．配列はデータを並べて，1つのデータにしたものである．JavaScriptでは配列は [と] で囲うことで作ることができる．Cの配列と異なり，違うデータ型のデータを１つの配列に入れることができる．

  例： `[1, 4, 3, 5. 6], [1, “山本”, 4, [4, 5]]`
- たとえば，2次元座標を表現することもできる．さらに3つの点を組にして三角形を表現することができる．

```
p1 = [1.0, 2.0]
p2 = [4.0, 6.6]
p3 = [6.0, 1.0]
pts = [p1, p2, p3]
```


配列のイメージ

配列 (array) はいくつかのデータを順序を付けて並べたもの．配列の要素はインデクス（数）によって参照することができる．




```
a = [3, 2, 6, 5, 8, 7, 9]
a[11] = 'abc'
puts(a[0])
puts(a[2])
puts(a[8])
puts(a[11])
puts(a)
```

```
3
6
undefined
abc
3,2,6,5,8,7,9,,,,,abc
```

未定義ということ

配列の生成と要素のアクセス

配列の生成：

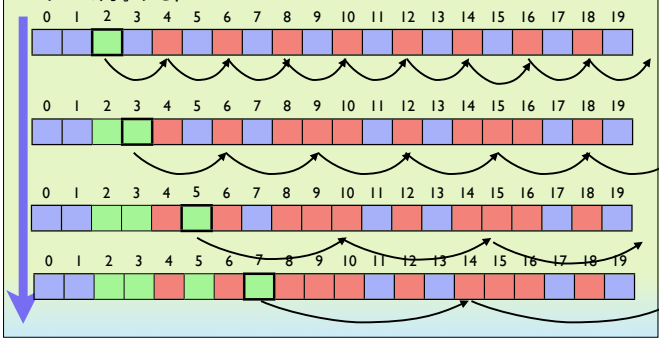
- `a = []` 空の配列を生成する
- `a = new Array(5)` 長さ5の配列を作る（オブジェクトとしての生成方法）
- `a = [4, 3, 5, 3]` 要素を指定して生成する

配列の要素へのアクセス：

- `a[23]` のようにインデックスを指定して参照する
- `a[23] = 55` のようにして内容を変化させることができる．

配列を利用して素数を求める (1)

アルゴリズム（エラトステネスのふるい）：ある整数n以下の素数をすべて列挙する.



配列を利用して素数を求める (2)

- 1000よりも小さな素数を求める.

```
var n = 1000
var a = []
var s = ''
for (var i = 2; i < n; i++) a[i] = 1
for (var i = 2; i < n; i++){
  if (a[i] == 1){
      for(var j = i * 2; j < n; j += i) a[j] = 0
          s += ' ' + i
  }
}

puts(s)
```

空の配列を作る

配列の中身を初期化する

素数の倍数を排除する

素数 i を記録する

配列を利用して素数を求める (3)

- 実行すると以下のように出力される.

```
 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61
67 71 73 79 83 89 97 101 103 107 109 113 127 131 137
139 149 151 157 163 167 173 179 181 191 193 197 199
211 223 227 229 233 239 241 251 257 263 269 271 277
281 283 293 307 311 313 317 331 337 347 349 353 359
367 373 379 383 389 397 401 409 419 421 431 433 439
443 449 457 461 463 467 479 487 491 499 503 509 521
523 541 547 557 563 569 571 577 587 593 599 601 607
613 617 619 631 641 643 647 653 659 661 673 677 683
691 701 709 719 727 733 739 743 751 757 761 769 773
787 797 809 811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941 947 953 967
971 977 983 991 997
```

プログラムはデータか？(1)

- 何も言わなければプログラムはデータではない.
- JavaScriptなどのプログラミング言語ではプログラムをデータにすることができる. データになったプログラムはその場では実行されない.

```
var x = 23
var y = 55
puts(x + y)
```

普通のプログラム

```
function (){
  var x = 23
  var y = 55
  puts(x + y)
}
```

データになったプログラム

プログラムはデータか？(2)

- プログラムはデータにすれば変数に代入することができる. **プログラムを実行するには，データの後に「()」をつける.** プログラムをデータにしたものを関数（関数オブジェクト）と呼ぶ.
【正確にはこのようなデータを**クロージャ** (closure；関数クロージャ）と呼ぶ】

変数oshigotoに関数オブジェクトを代入

```
var oshigoto = function (){
  var x = 23
  var y = 55
  puts(x + y)
}
oshigoto()
```

関数を実行

プログラムはデータか？(3)

関数オブジェクトを代入することを下のように書くことが許される

```
var oshigoto = function (){
  var x = 23
  var y = 55
  puts(x + y)
}
```

命令をカプセル化してデータにする

関数

```
var x = 23
var y = 55
puts(x + y)
```

oshigoto

この2つは同じ意味となる

```
function oshigoto(){
  var x = 23
  var y = 55
  puts(x + y)
}
```

引数と返り値

関数を作ったとき，中身の命令の列（プログラム）を実行するための情報を外から与えることができる．これを**引数 (argument)**と呼ぶ．また，計算した結果を関数の値として返すことができる．これを**返り値 (return value)**という．

```
var aisatu = function (name){
  return name + "さん，こんにちは。"
}
puts(aisatu("山本"))
puts(aisatu("加藤"))
```

引数 → name

返り値 →

関数 name 引数

```
return name + "さん，こんにちは。"
```

返り値（出力）

山本さん，こんにちは。
加藤さん，こんにちは。

関数と配列の比較

- JavaScriptでは関数と配列はそれほど違うものではない．

インデックス 0 1 2

[3, 6, 5]

データ

	機能	生成物の利用方法	メモ
配列	データを束ねてデータを作る．	束ねられたデータはa[i]のようにしてアクセスできる．また，a[i] = 10のようにして内容を書き換える	存在しないインデックスを指定してデータを代入すると配列は自動的に拡張される．
関数	命令を束ねてデータを作る．引数をつかって命令へデータを送れる．返り値によりデータを出力することができる	束ねられた命令はfoo(a, b, c) のようにして順に実行することができる．この場合，a, b, cは引数である．	命令を束ねて命令を作らないのは，命令にしてしまうと，即座に実行されてしまうからである．命令を束ねて命令を作るのは「複文」である．

```
function (a){ var b = a * a; return b}
```

引数　命令　命令

関数の例(1)--素数のプログラム--

- 素数を求めるプログラムを関数にしてみる．引数n未満の素数を計算して結果の文字列を返す．
- 関数にすると，引数の値を変えて実行することができる

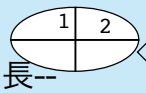
```
function prime(n){
  var a = []
  var s = []
  for (var i = 2; i < n; i++) a[i] = 1
  for (var i = 2; i < n; i++){
   if (a[i] == 1){
     for(var j = i * 2; j < n; j += i) a[j] = 0
     s.push(i)
   }
  }
  return s
}
puts(prime(100))
```

配列にpush(x) とメッセージを送るとxを最後に付加する

```
2,3,5,7,11,13,17,19,23,29
,31,37,41,43,47,53,59,61,
67,71,73,79,83,89,97
```

関数の例(2)--楕円の周長--

$$\ell = 4\sum_{i=0}^{2n-1}\sqrt{\left(\frac{1}{n}\right)^2+\left(\sqrt{1-\left(\frac{i}{2n}\right)^2}-\sqrt{1-\left(\frac{i+1}{2n}\right)^2}\right)^2}$$



- 前回の課題２を関数を用いて書いてみる．
- 関数の中に関数を書くと，そこだけで使える関数になる．

```
function ellipse(n){
  var foo = function(i){
    var x = i / 2 / n
    return Math.sqrt(1 - x * x)
  }
  var bar = function(x, y){
    return Math.sqrt(x * x + y * y)
  }
  var s = 0
  for (var i = 0; i < 2 * n; i++)
    s += bar(1/n, foo(i) - foo(i + 1))
  return 4 * s
}
puts(ellipse(1000))
```

$\sqrt{1-\left(\frac{i}{2n}\right)^2}$ の計算

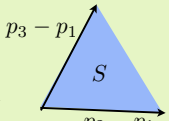
$\sqrt{x^2+y^2}$ の計算

関数の例(3)--三角形の面積--

- 平面上の3角形の面積を計算する．三角形の頂点は p =[x, y] のように配列による座標で表現する．

$$S = \frac{1}{2}\underline{|(p_3 - p_1) \times (p_2 - p_1)|}$$

ベクトルの外積




```
function vdiff(p1, p2){
   var x1 = p1[0]; var y1 = p1[1]
   var x2 = p2[0]; var y2 = p2[1]
   return [x2 - x1, y2 - y1]
}
function prod(p1, p2){
   var x1 = p1[0]; var y1 = p1[1]
   var x2 = p2[0]; var y2 = p2[1]
   return x1 * y2 - y1 * x2
}
function menseki(p1, p2, p3){
   return Math.abs(prod(vdiff(p3, p1), vdiff(p2,p1))) / 2
}
var p1 = [3, 3]
var p2 = [2, 1]
var p3 = [1, 7]
puts(menseki(p1, p2, p3))
```

ベクトルの引き算

p1, p2 → vdiff → p1 - p2

ベクトルの外積

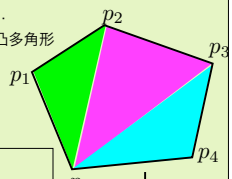
p1, p2 → prod → p1 × p2

p1, p2, p3 → menseki → s

三角形の面積の計算

関数の例(4)--凸多角形の面積--

- 凸多角形とは「へこみ」のない多角形のことである．
- 三角形の面積を計算する関数 mensekiがあれば，凸多角形の面積を計算する関数が作れる．
- n個の頂点からなる凸多角形を点の配列で与える．面積を計算するにはいくつかの三角形に分解して面積を足せば良い．



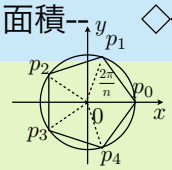

```
function polygon_menseki(poly){
   var n = poly.length
   var s = 0
   for (var i = 1; i < n - 1; i++)
     s += menseki(poly[0], poly[i], poly[i + 1])
   return s
}
```

polygon_menseki → 面積

mensekiが三角形の面積を計算するということだけ知っていればこの関数のプログラムを理解することができる．mensekiの中身は知らなくても問題ない．

関数の例(5)--正n角形の面積--

- 半径１の円周上に等間隔に点をとって，正n角形をつくり面積を計算する．




```
function regular_polygon_menseki(n){
  var poly = []
  for (var i = 0; i < n; i++){
   var theta = 2 * Math.PI * i / n
   var x = Math.cos(theta)
   var y = Math.sin(theta)
   poly.push([x, y])
  }
  return polygon_menseki(poly)
}

for (var i = 3; i < 20; i++){
  puts(i + ", " + regular_polygon_menseki(i))
}
```

```
3, 1.299038105676658
4, 2
5, 2.377641290737884
6, 2.598076211353316
7, 2.736410188638104
8, 2.82842712474619
9, 2.8925442435894273
10, 2.938926261462366
11, 2.973524496005786
12, 2.9999999999999996
13, 3.0207006182844958
14, 3.037186173822907
15, 3.050524823068502
16, 3.0614674589207183
17, 3.07055416259508004
18, 3.078181289931019
19, 3.0846449574444934
```

nが大きくなるとπに収束していく

関数内部で定義された変数について

関数内部でvarで定義された変数は**関数内部でしか利用できない変数（局所変数）**となる．varを使わないで使われる変数はJavaScriptシステム全体で**共通の変数（大域変数）**となる．

```
var a = 20
var b = 33
function foo(){
   var b = 30
   puts("a = " + a)
   puts("b = " + b)
}
foo()
puts("a = " + a)
puts("b = " + b)
```

大域変数 / 局所変数 → 違う変数

```
a = 20
b = 30
a = 20
b = 33
```

```
var a = 20
var b = 33
function foo(){
   b = 30
   puts("a = " + a)
   puts("b = " + b)
}
foo()
puts("a = " + a)
puts("b = " + b)
```

大域変数 / 大域変数 → 同じ変数

```
a = 20
b = 30
a = 20
b = 30
```


関数を使うと良いこと

- プログラムを適当な変数に結びつけておいて，後で何回でも使う．
- 引数を用いることによって，条件の異なる実行を何回でも行うことができる．
- プログラムを部品として扱うことができる（構造化プログラミング）．これは非常に重要である．人間は大きな（複雑な）プログラムを書くことが出来ないので，小さな（単純な）プログラムを繋げて大きなプログラムを作る．間違いの少ないプログラムを短時間で作るための方法は構造化プログラミングである．
- 関数はデータなので，それ自体を引数として与えることができる（これは次回以降の講義で説明する）．

大きな一塊 → 構造化 → 小さな塊の集合

再帰構造の例 -GNUとは何か？-

- GNUという言葉は，「ぐにゅー」と読む．GNUはUNIX互換のソフトウェアをすべてフリーウェアで作ろうとするプロジェクトの名前である．
- GNUはNHKのような頭字語である．GNUは以下のような言葉の頭字語になっている．

GNU = **G**NU is **N**ot **U**nix　自己参照

↑GNUの定義の中でGNUが使われている

GNU = GNU is Not Unix

= ‘GNU is Not Unix’ is Not Unix

= “GNU is Not Unix’ is Not Unix’ is Not Unix

= “‘GNU is Not Unix’ is Not Unix’ is Not Unix’ is Not Unix

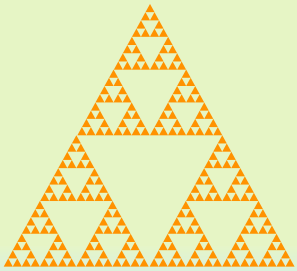
・・・

どこまでも続く

再帰構造の例 - シェルピンスキーガスケット - (1)

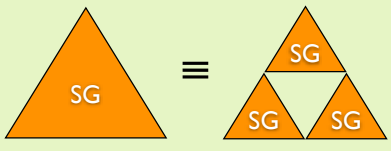
- シェルピンスキーガスケットはシェルピンスキー（1882-1969）によって考案された以下のような図形である．

Waclaw Sierpinski



再帰構造の例 - シェルピンスキーガスケット - (2)

- シェルピンスキーガスケット (SG) は以下のように定義される



SGの作り方（再帰的な説明）：

1. SGを１つ用意する
2. このSGのコピーを2つ用意する
3. それぞれを半分に縮小する
4. 3つの縮小されたSGを三角状に並べる

そもそもSGがないからSGを作るわけで最初からSGがあればSGは作らない…

鶏が先か卵が先か？

再帰構造の例 - シェルピンスキーガスケット - (3)

もともとシェルピンスキーガスケットを持っていないので，シェルピンスキーガスケットは作れない．

↓ ところが…

シェルピンスキーガスケットの近似を考えれば作ることができる！

SG(0) = T

SG(n) = SG(n-1) SG(n-1) SG(n-1)

SG(3)を作るにはSG(2)があればよい．
SG(2)を作るにはSG(1)があればよい．
SG(1)を作るにはSG(0)があればよい．
SG(0)は普通の三角形なので作れる

n→∞のときSG(n)→SGとなる．SGの近似を求めるアルゴリズムとなる．

再帰構造を用いると関数はさらに便利である

再帰構造を関数に採り入れると，複雑なことが信じられないくらい簡単に書けることがある．

1からnまでの和 ≡ 1からn-1までの和 + n
1から0までの和 ≡ 0

```
function sum_n(n){
  if (n == 0) return 0
  else return sum_n(n - 1) + n
}

puts(sum_n(100))
```

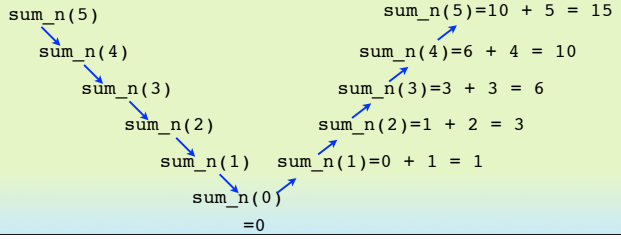
1から100までの和 → 5050

for文などの繰り返し構造を使わなくても計算できる！

関数 sum_n の動きの解析 (1)

- sum_nは順次自分自身を呼びながら，深く降りて行く．
- 最後まで降りるとそこから元の経路を辿って戻る．

```
function sum_n(n){
  if (n == 0) return 0
  else return sum_n(n - 1) + n
}

puts(sum_n(5))
```



関数 sum_n の動きの解析 (2)

実際にデバッグ用のコードを入れて実行させると以下のようになる

```
function sum_n(n){
  puts('-->sum_n(' + n + ')')
  if (n == 0) var val = 0
  else var val = sum_n(n - 1) + n
  puts('<--sum_n(' + n + ') = ' + val)
  return val
}

puts(sum_n(5))
```

再帰している部分

```
-->sum_n(5)
-->sum_n(4)
-->sum_n(3)
-->sum_n(2)
-->sum_n(1)
-->sum_n(0)
<--sum_n(0) = 0
<--sum_n(1) = 1
<--sum_n(2) = 3
<--sum_n(3) = 6
<--sum_n(4) = 10
<--sum_n(5) = 15
15
```


再帰呼び出しによる累乗の計算 (1)

- 再帰呼び出しを使うことで，足し算の繰返し表現できるので，累乗についても同様に計算することができる．

$$a^0 = 1$$
$$a^n = a^{n-1} \times a \quad (n >= 1)$$

このプログラムで問題なく計算できるが，1000乗だと1000回繰り返すことになる

```
function power(a, n){
  if (n == 0) return 1
  else return power(a, n - 1) * a
}

puts(power(2, 10))
```


再帰呼び出しによる累乗の計算 (2)

- もっと効率的なプログラムが書ける．

$$a^0 = 1$$
$$a^n = (a^{n/2})^2 \quad n\text{が偶数のとき}$$
$$a^n = (a^{(n-1)/2})^2 \times a \quad n\text{が奇数のとき}$$

```
function power(a, n){
  if (n == 0) return 1
  else if (n % 2 == 0){ /* nが偶数のとき */
    var b = power(a, n / 2);
    return b * b
  } else {              /* nが奇数のとき */
    var b = power(a, (n - 1) / 2)
    return b * b * a
  }
}

puts(power(2, 10))
```

再帰呼び出しによる累乗の計算 (3)

- なぜこのプログラムだと計算量が少ないのか？

power(a, 1000)
↓
power(a, 500)
↓
power(a, 250)
↓
power(a, 125)
↓
power(a, 125)
↓
power(a, 62)
↓
power(a, 31)
↓
power(a, 15)
↓
power(a, 7)
↓
power(a, 3)
↓
power(a, 1)
↓
power(a, 0) = 1

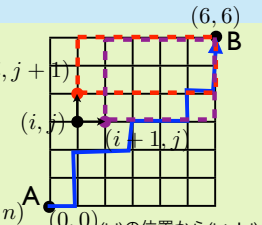
```
function power(a, n){
  if (n == 0) return 1
  else if (n % 2 == 0){ /* nが偶数のとき */
    var b = power(a, n / 2);
    return b * b
  } else {              /* nが奇数のとき */
    var b = power(a, (n - 1) / 2)
    return b * b * a
  }
}

puts(power(2, 10))
```

この程度の段数しか呼ばれない．1000段呼ばれる前のプログラムとはかなり状況が異なる

行き方の場合の数を数える (1)

6x6の格子状の迷路がある．AからBへ右への移動と上への移動のみを繰り返して移動する．このとき，すべての行き方は何通りあるか？

$$\varphi(m, j) = 1$$
$$\varphi(i, n) = 1$$
$$\varphi(i, j) = \varphi(i+1, j) + \varphi(i, j+1) \quad (i, j < n)$$



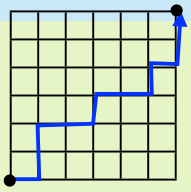

```
var m = 6
var n = 6
function phi(i, j){
   if (i == m) return 1
   else if (j == n) return 1
   else return phi(i + 1, j) + phi(i, j + 1)
}

puts(phi(0, 0))
```

(i, j)の位置から(i + 1, j)へ行く行き方と(i, j + 1)へ行く行き方はすべて異なる行き方である．それ以外に行き方はない．だから足すことで求まる

行き方の場合の数を数える (2)

このプログラムを実行すると以下のようになる．

```
var m = 6
var n = 6
function phi(i, j){
   if (i == m) return 1
   else if (j == n) return 1
   else return phi(i + 1, j) + phi(i, j + 1)
}

puts(phi(0, 0))
```



924

→↑↑→→↑→↑↑

このように経路を記述できる

別の求め方：

これは12個の場所のうち→を置く場所を6カ所決めるのと同じである．

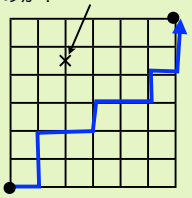
$${}_{12}C_6 = \left(\begin{array}{c} 12 \\ 6 \end{array} \right) = \frac{12 \times 11 \times 10 \times 9 \times 8 \times 7}{6 \times 5 \times 4 \times 3 \times 2 \times 1} = 924$$

行き方の場合の数を数える (3)

- もし工事中で通れない道があったらどうなるのか？

工事中

```
var m = 6
var n = 6
function phi(i, j){
   if (i == m) return 1
   else if (j == n) return 1
   else if (i == 2 && j == 4)
            return phi(i + 1, j)
   else return phi(i + 1, j) + phi(i, j + 1)
}
puts(phi(0, 0))
```



(2, 4)の位置では横にしか移動できない

849

別の求め方：

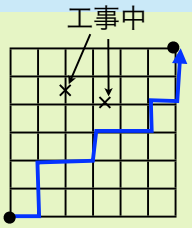
$$924 - \left(\begin{array}{c} 6 \\ 2 \end{array} \right) \times \left(\begin{array}{c} 5 \\ 1 \end{array} \right) = 924 - 15 \times 5 = 849$$

行き方の場合の数を数える (4)

もし2カ所工事中で通れない道があったらどうなるのか？

工事中

```
var m = 6
var n = 6
function phi(i, j){
   if (i == m) return 1
   else if (j == n) return 1
   else if (i == 2 && j == 4)
            return phi(i + 1, j)
   else if (i == 3 && j == 4)
            return phi(i, j+ 1)
   else return phi(i + 1, j) + phi(i, j + 1)
}
puts(phi(0, 0))
```



(3, 4)の位置では横にしか移動できない

639

まとめ

- 最初に配列の利用方法について説明した．配列を用いることにより，我々の身の回りの多くの事柄をプログラムで表現できるようになる．
- 関数の作り方について学んだ．関数を書くことによって，プログラムを部品化し，さらにそのプログラムをデータとして扱うことが可能になる．
- ある関数の中でその関数を呼び出すような形式のプログラムを再帰的な関数と呼ぶ．再帰的な関数を用いると複雑な事柄がとても単純に書けることがある．