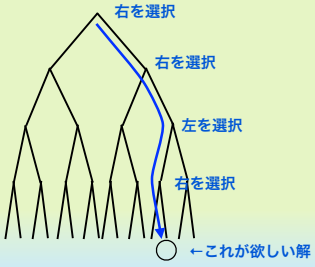
アルゴリズム・データ構造 I 第5回

２分法とニュートン法ー欲しい数値の見つけ方

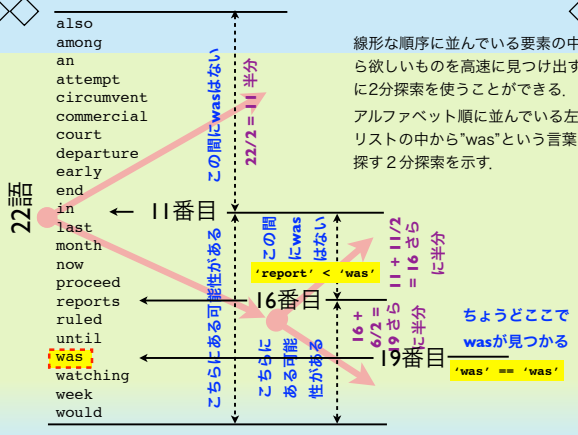
名城大学理工学部情報工学科
山本修身

２分探索法とは

- 以下のような木構造（これを2分木という）があるとする．この木構造において，それぞれのレベルで2つうちのどちらかを選択すると，最終的に欲しい解に辿り着くことができる場合，この探索方法を**2分探索法（バイナリーサーチ：binary search）**と呼ぶ．



2分探索の例（１）

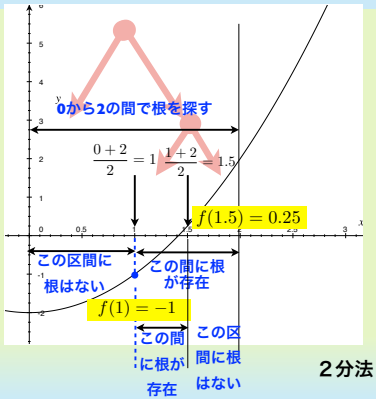


線形な順序に並んでいる要素の中から欲しいものを高速に見つけ出すのに2分探索を使うことができる．

アルファベット順に並んでいる左のリストの中から"was"という言葉を探す２分探索を示す．

２分探索の例（２）

- ２の平方根を計算してみる． $f(x) = x^2 - 2$ の根を求めれば良い



始めの状態：[0.0--2.0]
↓
次の状態：[1.0--2.0]
↓
その次の状態：[1.0-1.5]
↓
⋮

真の答えに近づく

近似解

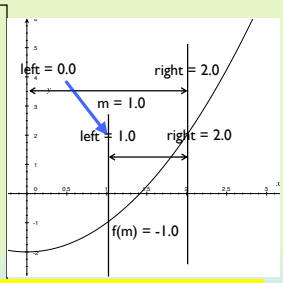
２分法 （bisection method）

実際に２の平方根を計算する

２分探索によって２の平方根を計算してみる．f(x) = x^2 - 2は x>0で単調に増加する．確実に√2が入っている区間[0, 2]を初期区間にする

```
function binary(){
   var EPS = 1.0e-6
   var left = 0.0
   var right = 2.0
   while (right - left > EPS){
      var m = (left + right) / 2
      var v = m * m - 2
      if (v < 0) left = m
      else right = m
   }
   puts(left + ", " + right)
}

binary()
```




```
1.4142131805419922, 1.4142141342163086
```

2の平方根を求めるとき何回繰り返しが起こるか (1)

- １回の繰り返しで区間の幅は確実に半分になる．

[left, right]
どちらに転んでも幅は半分になる
[left, (left + right) / 2]　　[(left + right)/2, right]

n回目の繰り返しにおける幅をw_nとすれば

$$w_n = \frac{w_0}{2^n} < \varepsilon$$

$$w_0 = |right - left|$$

となるとき停止する．この方程式を解けば，$\log_2 \frac{w_0}{\varepsilon} < n$

したがって，このプログラムは

$n = \left\lceil \log_2 \frac{w_0}{\varepsilon} \right\rceil$　ほど繰り返して停止することになる．

2の平方根を求めるとき何回繰り返しが起こるか (2)

- 1.0 x 10-6くらいまで|right - left|を小さくするには何回繰り返せばよいか？

$$\varepsilon = 1.0 \times 10^{-6}$$
$$w_0 = |right - left| = 2$$

$$n = \left\lceil \log_2 \frac{w_o}{\varepsilon} \right\rceil = \left\lceil \log_2 \frac{2}{1.0 \times 10^{-6}} \right\rceil$$
$$= \left\lceil \log_2 2 \times 10^6 \right\rceil = 1 + 6 \log_2 10 = 1 + 6\frac{1}{0.3} = 21 \text{回}$$

N 個の単語リストから目的の語を２分探索で見つけ出すための繰り返し回数

考え方は前述の方程式の根を求めるのと同じ．繰り返すごとに個数が半分ずつになっていくので，単語リストの大きさをN，繰り返し回数をnとしてとして

$$\frac{N}{2^n} \leq 1$$

となり，これを解けば，

$$\log_2 N \leq n$$

たとえば，100万語は行っているリストの中から目的の単語を見つけ出すのに，

$$\log_2 10^8 = 8 \log_2 10 = \frac{8}{0.3} = 26.67 < 27$$

の結果から27回繰り返せば見つかることになる．多くのデータから欲しいデータを見つけ出すのにこの方法が使えれば極めて高速に処理できる．

巨大な単語リストからの検索 (1)

- まずは端から順に比べて探す方法でやってみる．

```
function find(word, wl){
   for (var i = 0; i < wl.length; i++){
      if (word == wl[i] ){
         return [i, word]
         break;
      }
   }
   return null
}

var t1 = new Date()
res = find("zygote", wordlist)
var t2 = new Date()
puts(t2 - t1)
puts(res)
```

「単語リスト付き」のプログラミング環境を用いること

2ミリ秒で見つかった．この単語は115515番目で見つかった．リストのほぼ最後の単語である．

```
2
115515,zygote
```

巨大な単語リストからの検索 (2)

- バイナリーサーチによる探索

実際の検索に要する時間は相当の違いがあると考えられるが時間計測の解像度があまり良くないので，うまく比較できていない

```
function bsearch(p, q, word, wl){
   while (q - p > 1){
     var m = Math.floor((p + q) / 2)
     if (wl[m] == word) return [m, word]
     else if (wl[m] < word) p = m
     else q = m
   }
   return [null, word]
}

var t1 = new Date()
res = bsearch(0, wordlist.length, "zygote", wordlist)
var t2 = new Date()
puts(t2 - t1)
puts(res)
```

1ミリ秒未満で見つかった．

```
0
115515,zygote
```

巨大な単語リストからの検索 (3)

- 同じことを1000回ずつ繰り返して時間を測る．その結果1000倍近く実行時間が異なることがわかる．

```
var t1 = new Date()
  for (var i = 0; i < 1000; i++)
    res = find("zygote", wordlist)
var t2 = new Date()
puts("dumb algorithm -> " + (t2 - t1) + " ms")
puts(res)
```

端から探す方法

```
var t1 = new Date()
  for (var i = 0; i < 1000; i++)
    res = bsearch(0, wordlist.length, "zygote", wordlist)
var t2 = new Date()
puts("binary search algorithm -> " + (t2 - t1) + " ms")
puts(res)
```

バイナリーサーチ

```
dumb algorithm -> 1854 ms
115515,zygote
binary search algorithm -> 2 ms
115515,zygote
```

バイナリーサーチを再帰で書く

- バイナリーサーチは再帰で書いた方が自然である．ただし，効率の問題は別である．

```
function binary(left, right){
   var EPS = 1.0e-6
   if (right - left <= EPS){
      puts(left + ", " + right)
   } else {
      var m = (left + right) / 2
      if (m * m - 2 < 0) binary(m, right)
      else binary(left, m)
   }
}

binary(0, 2)
```

再帰バージョン

特にこのような形の再帰呼び出しは再帰呼び出しから返ったあと，何も処理する必要がないもので末尾再帰 (tail recursion) とよばれている．

```
function binary(){
   var EPS = 1.0e-6
   var left = 0.0
   var right = 2.0
   while (right - left > EPS){
      var m = (left + right) / 2
      if (m * m - 2 < 0) left = m
      else right = m
   }
   puts(left + ", " + right)
}

binary()
```

繰り返しバージョン

より利用可能範囲の広い探索

バイナリサーチは確実に探索範囲を半分にしながら計算が進行するので，一定回数で解に到達することができる．これに対して，そこまでの保証のない探索（近似）を考えると，適用範囲を広くことができる．

求めたい解 を x としたとき， x = F(x) という性質を満たす解を求めたい場合，

$$x_{n+1} = F(x_n) \qquad \lim_{n\to\infty} x_n = x$$

という反復を繰り返すことによって，求めることができる場合がある．このとき，この反復によって解が求まるには解の近傍で以下の条件が成り立つことが必要．

$$\left|\frac{\partial F(x)}{\partial x}\right| < 1$$

反復法を用いた平方根の計算 (1)

- √n の計算を考える．ただし，n = 2, 3, 4, … とする．

$$x^2 = n \longrightarrow x^2 + x = n + x \longrightarrow x = \frac{x+n}{x+1}$$

$$F(x) = \frac{x+n}{x+1} \quad \text{とおくと} \quad F'(x) = -\frac{n-1}{(x+1)^2}$$

$$|F'(\sqrt{n})| = \left|\frac{n-1}{(\sqrt{n}+1)^2}\right| < \frac{n}{n} = 1$$

よって，F(x)による反復は√nの近傍で真の解に収束する．

反復法を用いた平方根の計算 (2)

以下のようなプログラムによって整数の平方根を反復法で計算できる

```
function iter(n){
    var EPS = 1.0e-7
    function F(x){
        return (x + n) / (x + 1)
    }
    var x = 2.0
    var x2 = 0.0
    while (Math.abs(x - x2) > EPS){
        x2 = x
        x = F(x)
    }
    return x
}
```

```
for (var i = 2; i < 20; i++)
   puts(iter(i))
```

```
1.4142135731001355
1.7320507984408398
2
2.236067997043607
2.4494897282044055
2.6457512925597193
2.828427105497636
2.9999999821186067
3.1622776899651837
3.316624814188631
3.464101581532335
3.6055512504153806
3.7416574187451346
3.872983369199965
3.9999999724948676
4.12310565747641
4.242640651201505
4.358898918732829
```


反復法を用いた連立一次方程式の解法 (1)

以下のような連立1次方程式を解く．　$\overline{x} = A^{-1}b$

$$Ax = b$$

$A = D + U + L$ として，おけば，$Dx = b - (U + L)x$

となるので，　$x = D^{-1}(b - (U + L)x)$ 従って，

$F(x) = D^{-1}(b - (U + L)x)$　と定義すれば，

反復　$x_{n+1} = F(x_n)$　によって解を求めることができる．

ただし，収束するには条件 $\left|\det D^{-1}(U + L)\right| < 1$

が成り立たないといけない．この方法は**ヤコビ法**と呼ばれるものである．

ただし，Dは対角行列，UとLはそれぞれ上と下三角行列

反復法を用いた連立一次方程式の解法 (2)

- 以下のような方程式を解いてみる．

$$\begin{pmatrix} 1 & 2 \\ 1 & 4 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

これに反復法を適用してみる．

```
function jacobi(n){
    var a = 1, b = 2, c = 1, d = 4
    var e = 2, f = -3
    function F(p){
        var [x, y] = p
        return [(e - b * y) / a, (f - c * x) / d]
    }
    var p = [0, 0]
    for (var i = 0; i < n; i++){
        p = F(p)
    }
    return p
}
```

```
for (i = 0; i < 10; i++)
    puts(i * 5 + " " + jacobi(i * 5))
```

```
0 0,0
5 5.75,-2.0625
10 6.78125,-2.421875
15 6.9609375,-2.486328125
20 6.9931640625,-2.49755859375
25 6.998779296875,-2.49957275390625
30 6.999786376953125,-2.4999237060546875
35 6.999961853027344,-2.4999866485595703
40 6.999993324279785,-2.499997615814209
45 6.9999988079071045,-2.499999582674866
```


ニュートン法について

ニュートン法は探索アルゴリズムではなく，数値を近似するためのアルゴリズムである．しかし，区分２分法で方程式の根を求めるなどの場合には，区分２分法と同じように動く．また，区間という概念を必要としないので，2次元以上の問題を解くこともできる．

$x' = x + \varepsilon$ とおく

ニュートン法はある近似値xからそれよりも良い近似値x'を計算する方法である．

$$f(x') = f(x + \varepsilon) = f(x) + f'(x)\varepsilon + o(\varepsilon) = 0$$

$$\varepsilon = -f(x)/f'(x) \longrightarrow \boxed{x' = x - f(x)/f'(x)}$$

◇◇ニュートン法で√2を計算してみる (1)◇◇

- √2を計算するということは，f(x) = x^2 - 2 の根を求めることである．すなわち，

$$f(x) = x^2 - 2$$
$$f'(x) = 2x$$

反復公式

$$x' = x - \frac{x^2 - 2}{2x} = \frac{2x^2 - x^2 + 2}{2x} = \frac{x^2 + 2}{2x}$$

$$x' = x - f(x)/f'(x)$$

◇◇ニュートン法で√2を計算してみる (2)◇◇

実際のニュートン法のプログラムは以下のとおり．

ニュートン法の収束は非常に高速である（2次収束）

区分２分法の収束は1次収束

```
function newton2(){
   var x = 2.0
   for (var i = 0; i < 10; i++){
      x = (x * x + 2) / x / 2
      puts(i + " " + x)
   }
}

newton2()
```

```
0 1.5
1 1.4166666666666667
2 1.4142156862745099
3 1.4142135623746899
4 1.414213562373095
5 1.414213562373095
6 1.414213562373095
7 1.414213562373095
8 1.414213562373095
9 1.414213562373095
```

4回目で収束している

◇◇Newton法の近似過程をグラフにする◇◇

```
pt1 = [0.1, 0.1]
pt2 = [0.8, 0.8]
N = 100

function conv(i){
   var x = i / N * 2.4
   var y = x * x - 2
   return convpt([x, y])
}
function convpt(pt){
   var x = pt[0]
   var y = pt[1]
   var xx = x / 2.4 / 2 + 0.5
   var yy = y / 2.4 / 2 + 0.5
   return [xx, yy]
}
function draw_graph(){
   draw_line([0, 0.5], [1, 0.5])
   draw_line([0.5, 0], [0.5, 1])

   for (i = -N - 1; i < N; i++){
      pt1 = conv(i)
      pt2 = conv(i + 1)
      draw_line(pt1, pt2)
   }
}
```

```
function f(x){return x * x - 2}
function fd(x){return 2 * x}

function draw_newton(){
  var x = 0.5
  draw_line(convpt([x, 0]), convpt([x, f(x)]))
  for (var i = 0; i < 5; i++){
   var x_old = x
   var y = f(x_old)
   var x = x - f(x) / fd(x)
   var pt1 = convpt([x_old, y])
   var pt2 = convpt([x, 0])
   var pt3 = convpt([x, f(x)])
   draw_line(pt1, pt2)
   draw_line(pt2, pt3)
   puts(x)
  }
}
draw_graph()
draw_newton()
```


◇◇ 実行結果 ◇◇

- 前のスライドのプログラムを実行したところ