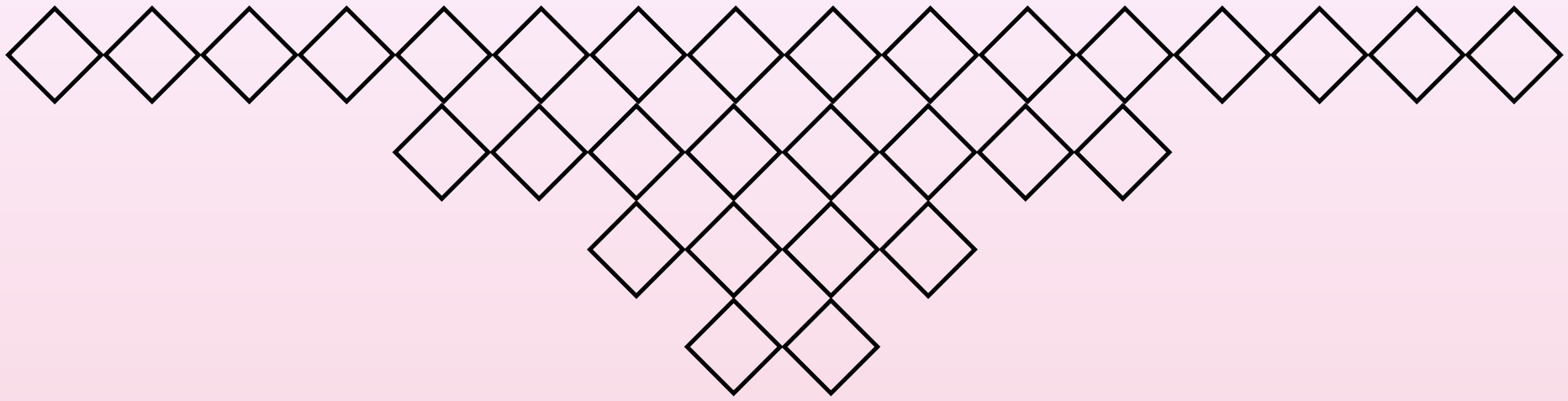
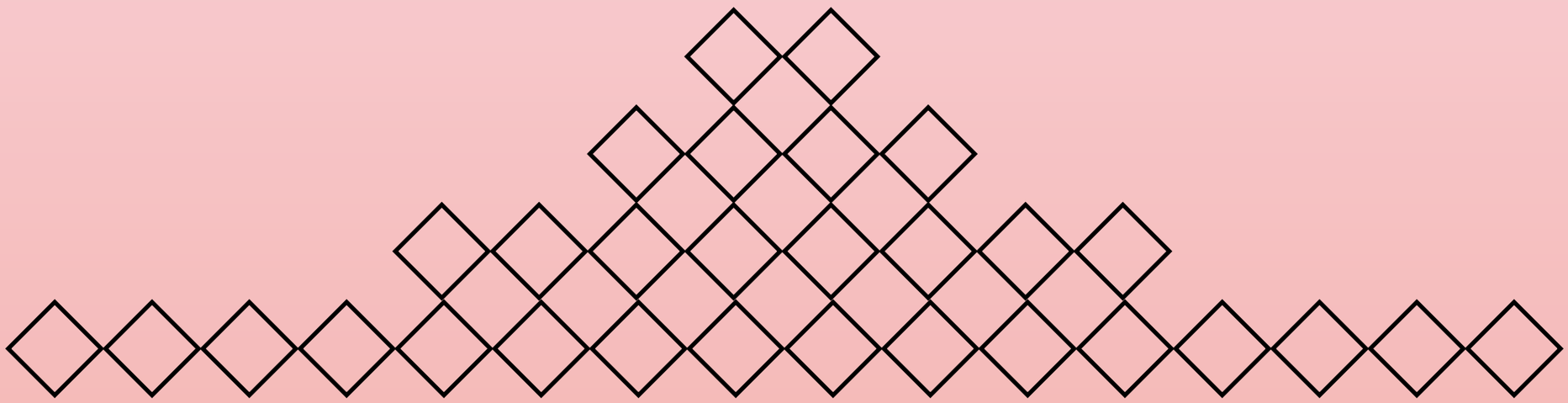


# アルゴリズム・データ構造 I 第13回

# 整列に要する計算時間

名城大学理工学部情報工学科

山本修身



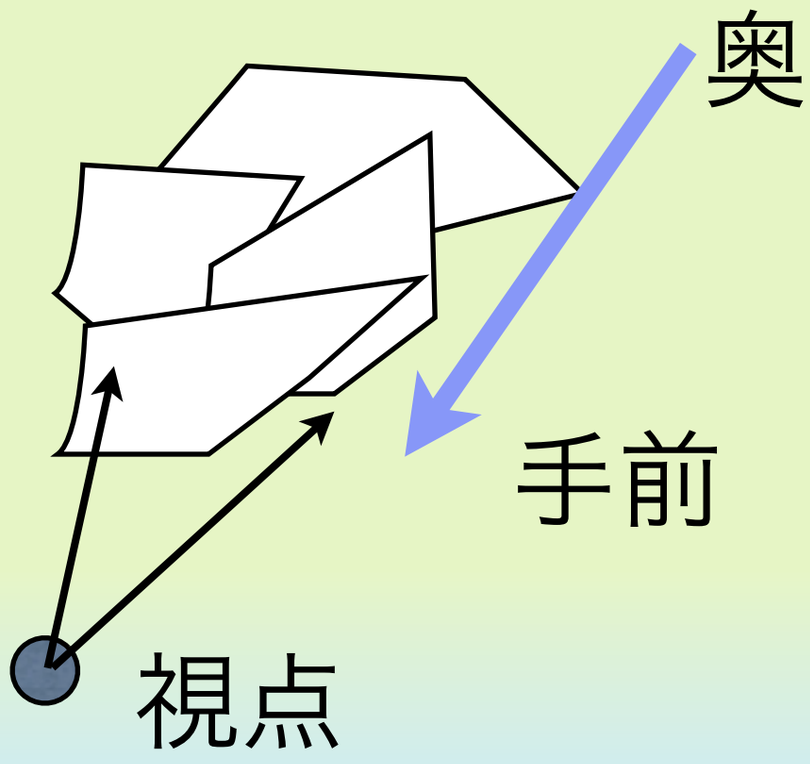
# ◇◇　これまでに出て来たソーティングアルゴリズム　◇◇

- これまでいくつかのアルゴリズムがでてきた.

1.セレクションソート（遅い）
2.クイックソート
3.マージソート
4.ヒープソート
5.コムソート（遅そうに見えて意外に速い．場合によってはクイックソートよりも高速．）

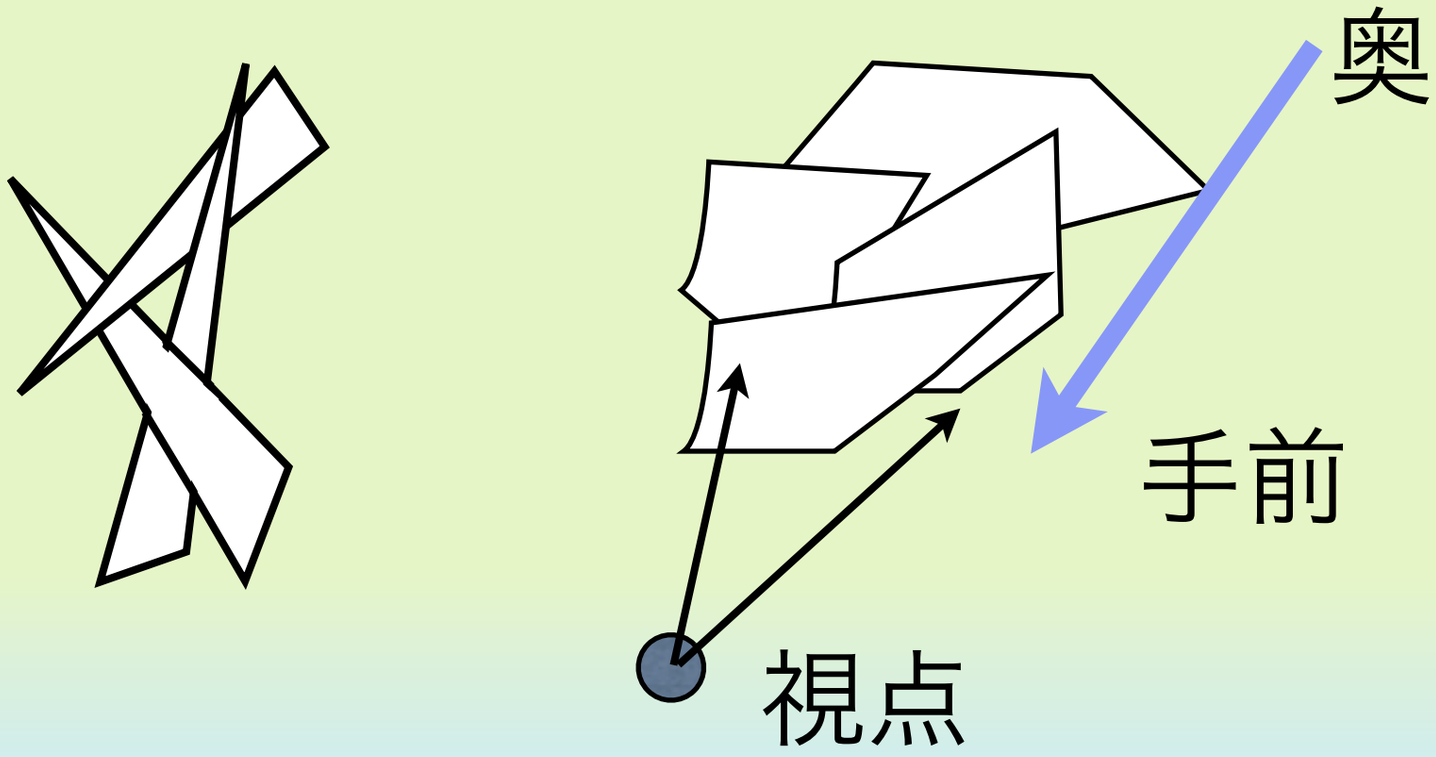
# ソーティングの応用例：ペインターズアルゴリズム (1)

- 図形をディスプレイ上に描画する最も簡単なアルゴリズムはペインターズアルゴリズム (painter's algorithm)である.
- 描画対象はポリゴン（多角形）の集合であるとする．ポリゴンがポリゴンを隠すので，すべてのポリゴンを描画すると正しいイメージが得られない．そこで，奥にあるポリゴンが順に前に向かって描画するのがこのアルゴリズムである.



# ソーティングの応用例：ペインターズアルゴリズム (2)

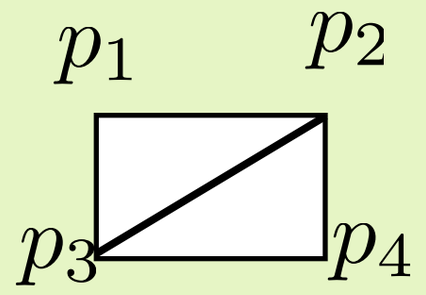
- ここではポリゴンとして最も単純な三角形のみを考える.
- 後ろの三角形から順に描画すれば良いが，左下のようなすくみの状態になっていると，うまく描画することができない．このようなケースはより細かく三角形に分割すればよい.

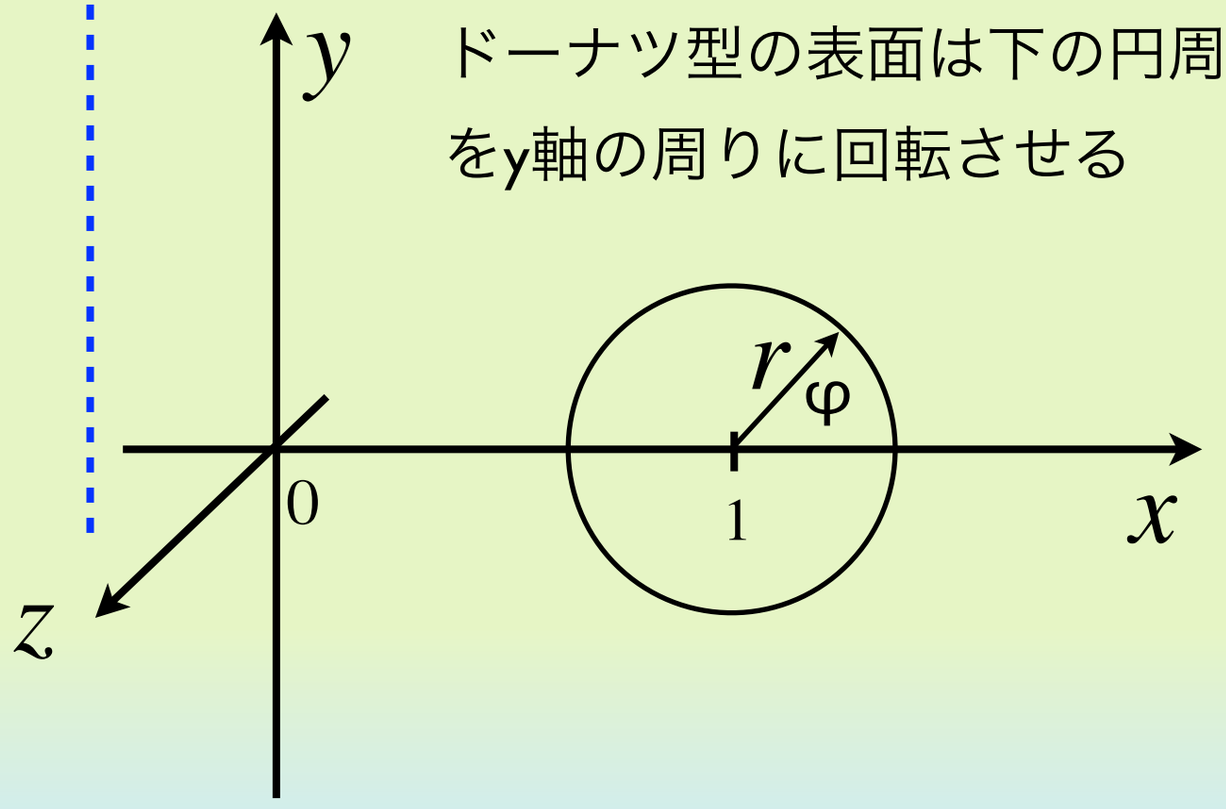


# ソーティングの応用例：
# ペインターズアルゴリズム (3)

まず，図形（ここではドーナツ型）をポリゴンの集合表現する.

```
function make_polygons(m, n){
   polygons = []
   var pi = Math.PI
   var r = 0.4
   for (var i = 0; i < m; i++){
     var th1 = 2 * pi * i / m
     var th2 = 2 * pi * ((i + 1) % m) / m
     for (var j = 0; j < n; j++){
        var phi1 = 2 * pi * j / n
        var phi2 = 2 * pi * ((j + 1) % n) / n
var x1 = r * Math.cos(phi1) + 1
var y1 = r * Math.sin(phi1)
var x2 = r * Math.cos(phi2) + 1
var y2 = r * Math.sin(phi2)
var xx11 = x1 * Math.cos(th1)
var zz11 = x1 * Math.sin(th1)
var xx12 = x1 * Math.cos(th2)
var zz12 = x1 * Math.sin(th2)
var xx21 = x2 * Math.cos(th1)
var zz21 = x2 * Math.sin(th1)
var xx22 = x2 * Math.cos(th2)
var zz22 = x2 * Math.sin(th2)
var p1 = [xx11, y1, zz11]
var p2 = [xx12, y1, zz12]
var p3 = [xx21, y2, zz21]
var p4 = [xx22, y2, zz22]
        polygons.push([p1, p2, p3])
        polygons.push([p2, p3, p4])
     }
   }
   return polygons
}
```



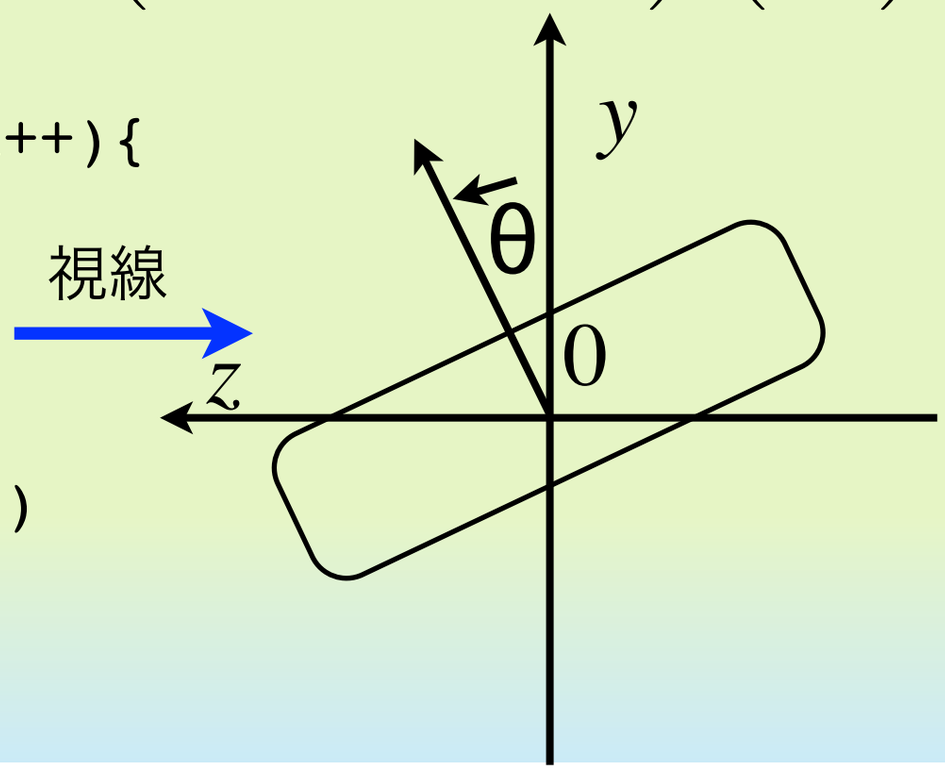



# ソーティングの応用例：ペインターズアルゴリズム (4)

```
function rot_yz(polygons){
    var rr = function(pt){
        var th = 0.5
        var x = pt[0]
        var y = pt[1]
        var z = pt[2]
        var c = Math.cos(th)
        var s = Math.sin(th)
        var yy = y * c - z * s
        var zz = y * s + z * c
        return [x, yy, zz]
    }
    var pp = []
    for (var i = 0; i < polygons.length; i++){
        var p = polygons[i]
        var pt1 = p[0]
        var pt2 = p[1]
        var pt3 = p[2]
        pp.push([rr(pt1), rr(pt2), rr(pt3)])
    }
    return pp
}
```

与えられた三角形の集合をそれぞれ原点を中心に同じ方向に回転させる．x座標は変化させないで，y-z平面上で回転させる．

$$\begin{pmatrix} y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix}$$



# ソーティングの応用例：ペインターズアルゴリズム (5)

それぞれの三角形について，その重心のz座標を深さと考えて，深さについて三角形をソートする．ここではとりあえずセレクションソートを用いる．

```
function sort_p(polygons){
    for (var i = 0; i < polygons.length - 1; i++){
      var mi = i
      var mv = center_z(polygons[i])
      for (var j = i + 1; j < polygons.length; j++){
        if (mv > center_z(polygons[j])){
          mi = j
          mv = center_z(polygons[j])
        }
      }
      var tmp = polygons[mi]
      polygons[mi] = polygons[i]
      polygons[i] = tmp
    }
}
```

```
function center_z(polygon){
    var p1 = polygon[0]
    var p2 = polygon[1]
    var p3 = polygon[2]
    var z1 = p1[2]
    var z2 = p2[2]
    var z3 = p3[2]
    return (z1 + z2 + z3) / 3
}
```

# ソーティングの応用例：ペインターズアルゴリズム (5)

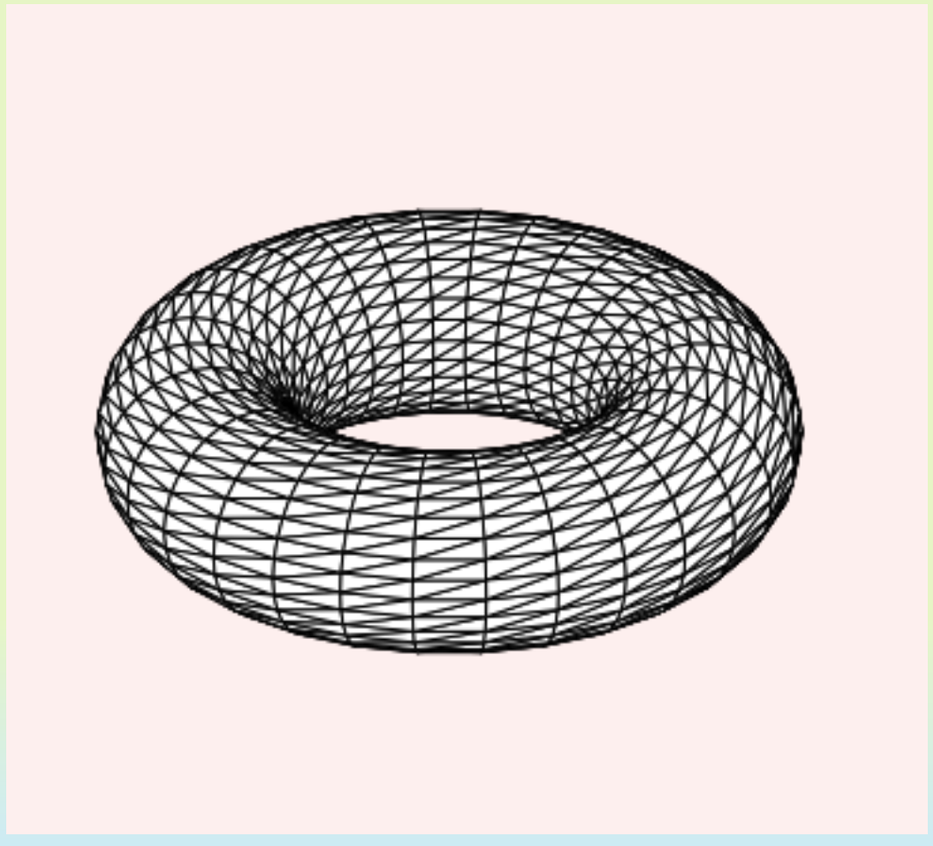
```
function draw_tr(pt1, pt2, pt3){  ←内部を白くした三角形を描画する
    draw_triangle_w(pt1, pt2, pt3)
    draw_line(pt1, pt2)
    draw_line(pt2, pt3)
    draw_line(pt3, pt1)
}
function pt3_2(pt){    ←深さ成分であるz座標を捨てて，(0, 0)-(1,
     var x = pt[0]     1)のフィットさせるための座標を計算する
     var y = pt[1]
     return [x / 4 + 0.5, y / 4 + 0.5]
}

function draw_polygons(polygons){    ←実際に三角形の列について描画する.
    for (var i = 0; i < polygons.length; i++){
      var p = polygons[i]
      var pt1 = p[0]
      var pt2 = p[1]
      var pt3 = p[2]
      draw_tr(pt3_2(pt1), pt3_2(pt2), pt3_2(pt3))
    }
}
```

ソートした後，実際に描画する.

# ソーティングの応用例：ペインターズアルゴリズム (6)

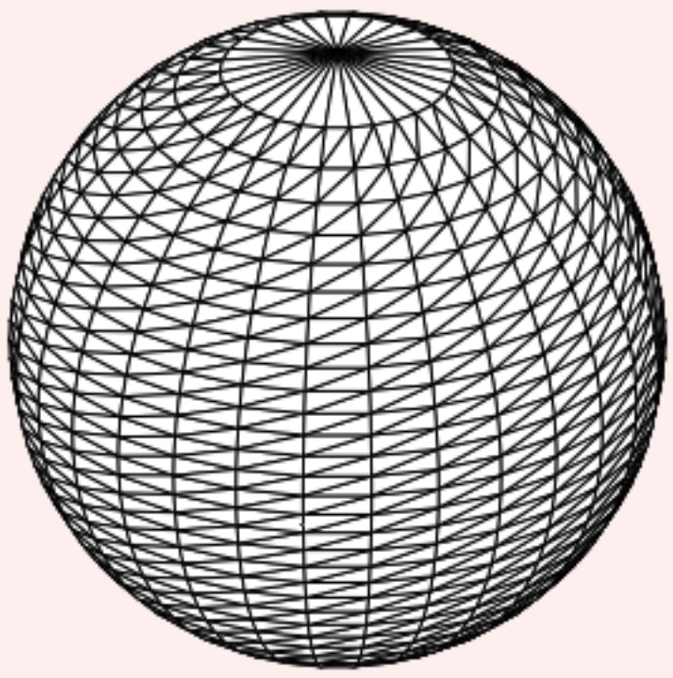
- 実際に動かすプログラムは以下のとおり

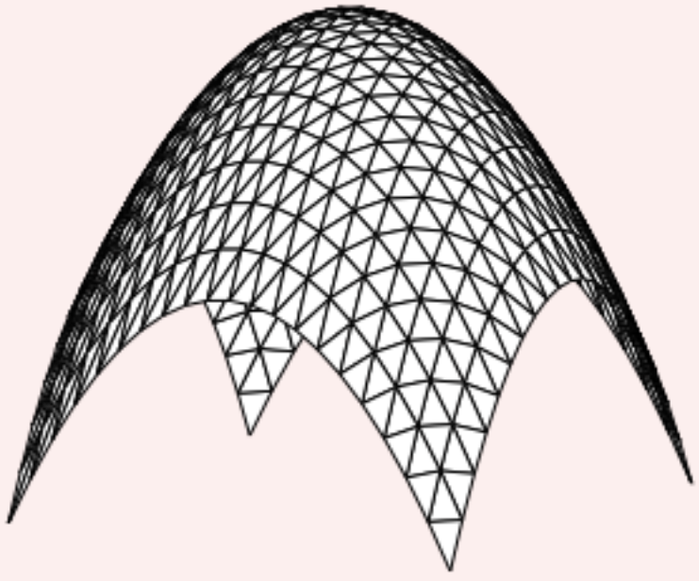
```
var polygons = rot_yz(make_polygons(30, 30))
sort_p(polygons)
draw_polygons(polygons)
```



# ソーティングの応用例：ペインターズアルゴリズム (7)

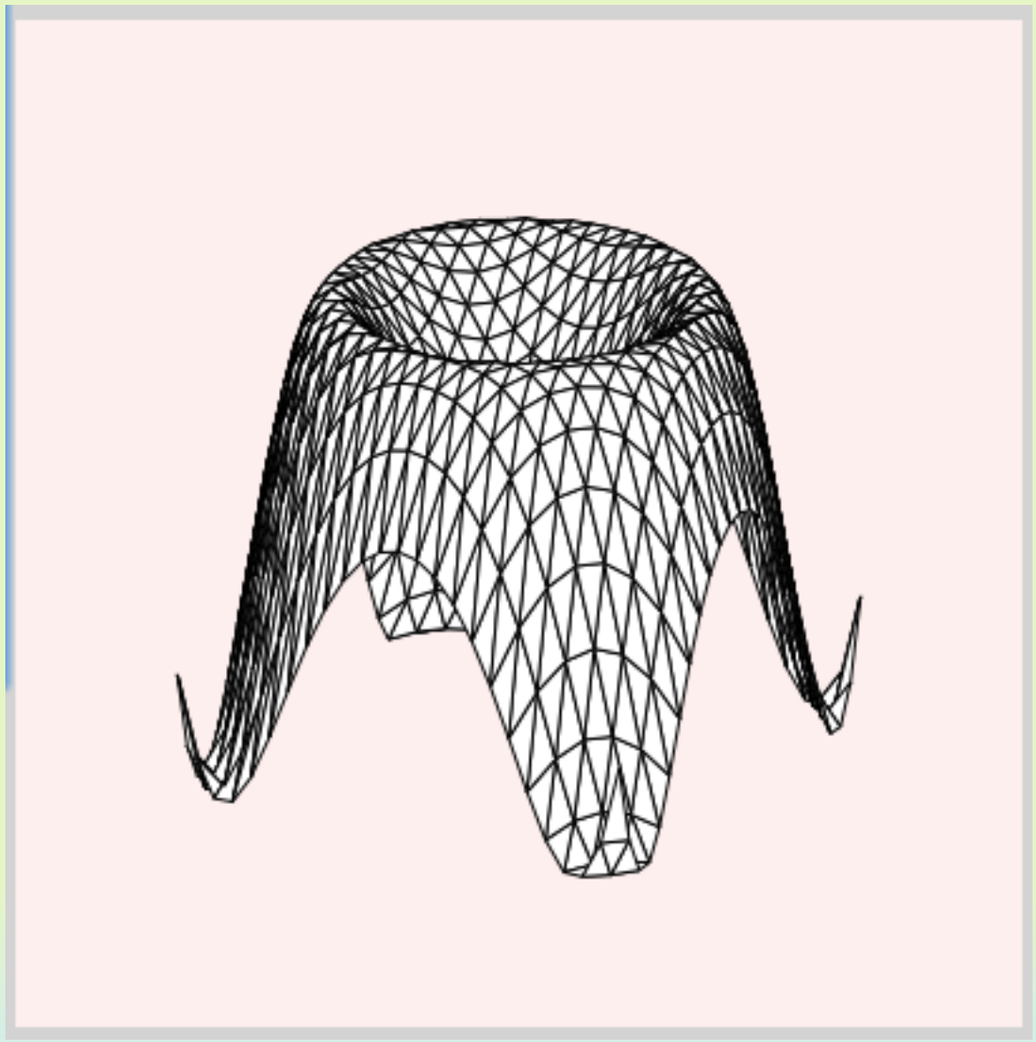
プログラムを多少変形すると色々な図形を描画することができる





# ソーティングの応用例：ペインターズアルゴリズム (8)

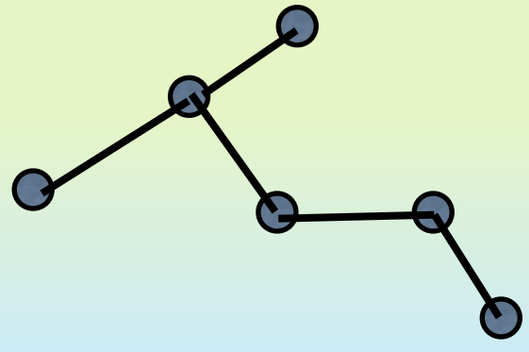
- ポリゴンをどのように配置するかによる（ソーティングとは関係ないが）



# ソーティングの応用例：
# 最小木で点どうしを結ぶ (1)

平面上にn個の点が与えられたとき，点同士を結んですべての点が連結になるようにして，なおかつ繋いだ長さの総和を最少にするにはどうしたらよいか？このように点を結んだグラフを最小木 (minimum spanning tree) と呼ぶ．このように結ぶ方法として知られているものにKruskal のアルゴリズムがある（アルゴリズムの正当性については ここでは述べない）．

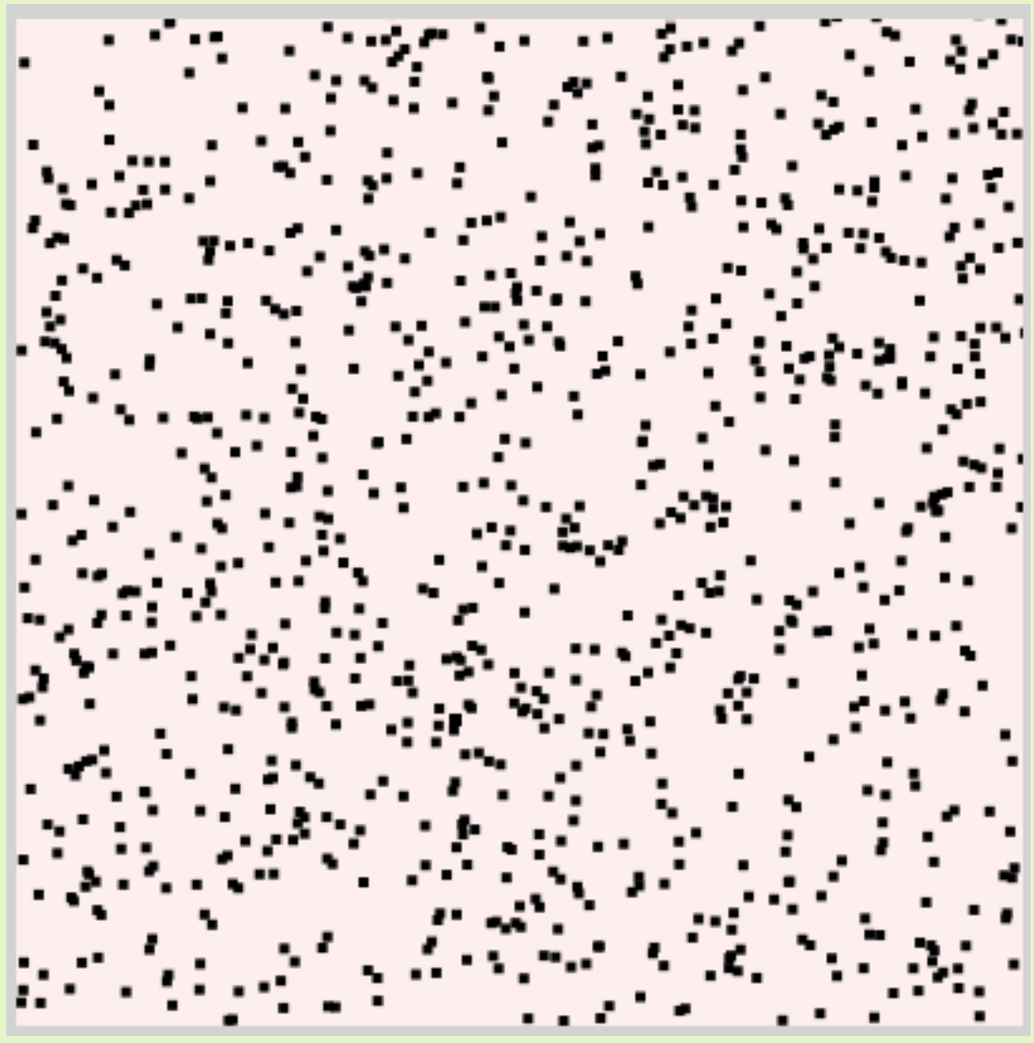
アルゴリズムの概略は以下のとおり： まず，とりうる枝をすべて列挙して，それを長さの小さい順にソートする．短い枝から順に取り出して，その枝を付け加えたときループが出来なければ，最小木の枝とする．全体が連結になるまでこれを繰り返す．



# ソーティングの応用例：最小木で点どうしを結ぶ (2)

- まずは，平面上にランダムに1000個の点を発生させる

```
function make_points(n){
    var pts = []
    for (var i = 0; i < n; i++){
        var x = Math.random()
        var y = Math.random()
        pts.push([x, y])
    }
    return pts
}


function draw_points(lst){
    dx = 0.005
    dy = 0.005
    for (var i = 0; i < lst.length; i++){
        var x = lst[i][0]
        var y = lst[i][1]
        draw_rect([x-dx, y-dy], [x + dx, y + dy])
    }
}

draw_points(make_points(1000))
```



# ソーティングの応用例：最小木で点どうしを結ぶ (3)

- 点のリストから辺のリストを作る.
  辺は [長さ，点番号１，点番号２] という形になっている

```
function make_edge_list(lst){
    function sqr(x){ return x * x }
    function dist2(p1, p2){
        var [x1, y1] = p1
        var [x2, y2] = p2
        return sqr(x1 - x2)  + sqr(y1 - y2)
    }
    var elist = []
    for (var i = 0; i < lst.length; i++){
        var p1 = lst[i]
        for (var j = i + 1; j < lst.length; j++){
            var p2 = lst[j]
            var d = dist2(p1, p2)
            if (d < 0.04){
                elist.push([d, i, j])
            }
        }
    }
    return elist
}
```

# ソーティングの応用例：最小木で点どうしを結ぶ (4)

- 辺の長さの短い順にソートする.

```
function sort_edge_list(elist){
    if (elist.length <= 1) return elist
    var key = elist[0][0]
    var low = []
    var eq = []
    var high = []
    for (var i = 0; i < elist.length; i++){
        if (elist[i][0] < key) low.push(elist[i])
        else if (elist[i][0] > key) high.push(elist[i])
        else eq.push(elist[i])
    }
    return sort_edge_list(low).concat(eq.concat(sort_edge_list(high)))
}
```

```
a = make_edge_list(make_points(1000))
a = sort_edge_list(a)
puts(a.length)
puts(a[0])
puts(a[999])
```

```
51675
5.925218054709946e-7,386,638
0.0006871094447426732,406,757
```

# ソーティングの応用例：最小木で点どうしを結ぶ (5)

- 最小木を構成する．すでに繋がっている点どうしはgdadの値が同じになるようにする．

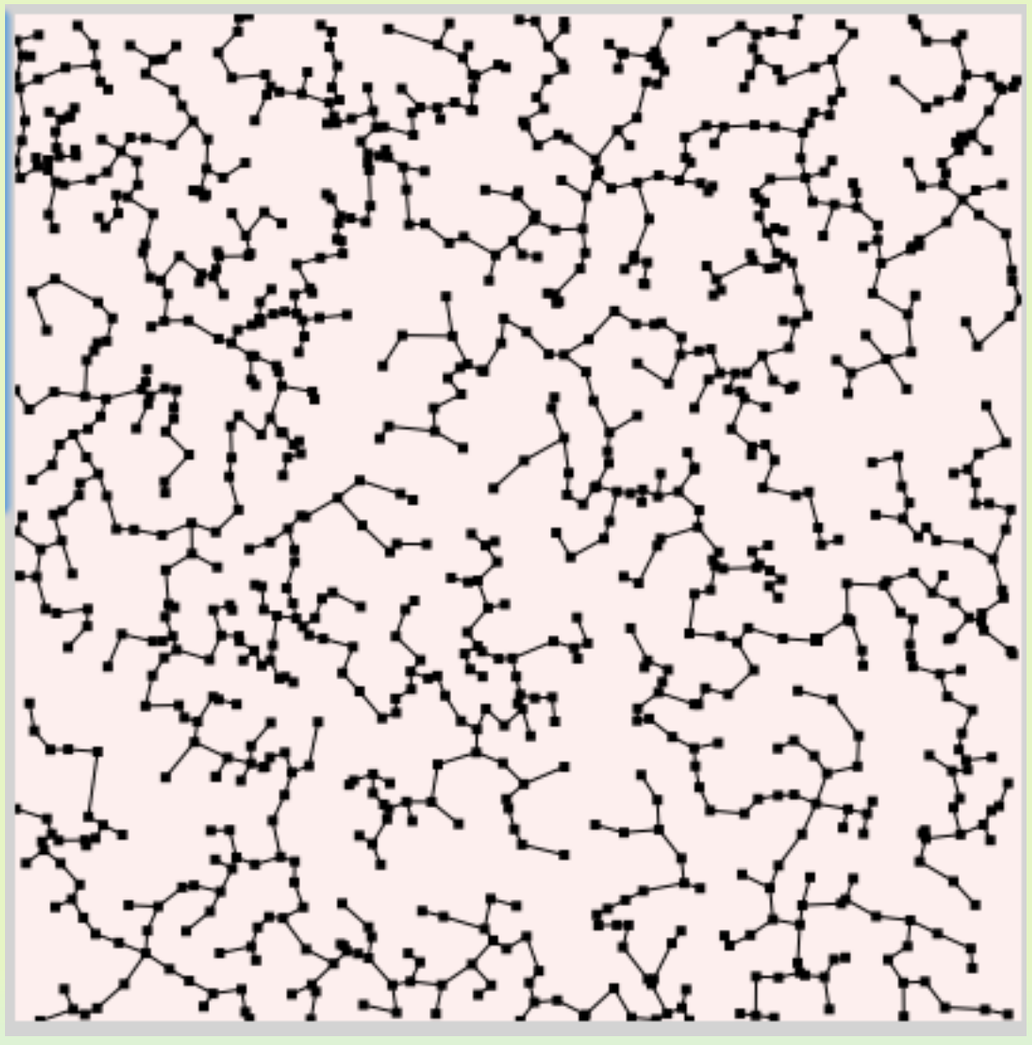
```
function make_MST(slist, n){
    var dad = []
    for (var i = 0; i < n; i++){
        dad[i] = -1
    }
    function gdad(i){
        while (dad[i] != -1){
            i = dad[i]
        }
        return i
    }
    var k = 0;
    var count = 0
    var res = []
    while (true){
        var [dist, ix, iy] = slist[k]
        if (gdad(ix) != gdad(iy)){
            dad[gdad(ix)] = gdad(iy)
            res.push([ix, iy])
            count += 1
        }
        if (count >= n - 1) break
        k += 1
    }
    return res
}
```

```
n = 1000
pts = make_points(n)
a = make_edge_list(pts)
a = sort_edge_list(a)
puts(a.length)
puts(a[0])
puts(a[n - 1])
mst = make_MST(a, n)
for (var i = 0; i < 10; i++)
   puts(mst[i])
```

```
25,522
39,969
234,581
236,703
229,642
117,938
334,699
817,974
802,887
660,966
```

# ソーティングの応用例：最小木で点どうしを結ぶ (6)

- 結果を表示する.



```
function draw_edges(mst, pts){
    for (var i = 0; i < mst.length; i++){
      var [i1, i2] = mst[i]
      var pt1 = pts[i1]
      var pt2 = pts[i2]
      draw_line(pt1, pt2)
    }
}
```

```
n = 1000
pts = make_points(n)
draw_points(pts)
a = make_edge_list(pts)
a = sort_edge_list(a)
puts(a.length)
puts(a[0])
puts(a[n - 1])
mst = make_MST(a, n)
draw_edges(mst, pts)
```

# ◇◇ ソーティングに要する計算時間 (1) ◇◇

- いくつかのソーティングアルゴリズムについて考えたが，それぞれのアルゴリズムで計算時間を見積もることができる．ソーティングの場合，「比較する」という動作が計算のほとんどであるので，比較の回数の総和を計算量と考える．
- まず，セレクションソートについて考えてみる．n個の要素をソートする場合，最初，n個の中の最小値を探す．これに要する比較回数は n - 1回である．その後，要素を入れ換えて，残りの n - 1個の要素の中の最小値を探す．これを残りが2個なるまで繰り返す．比較回数の総和は

$$T = \sum_{i=1}^{n-1}(n-i) = n(n-1) - \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$$

# ◇◇ ソーティングに要する計算時間 (2) ◇◇

- 計算時間が実際にどのくらいになるのかやってみる.

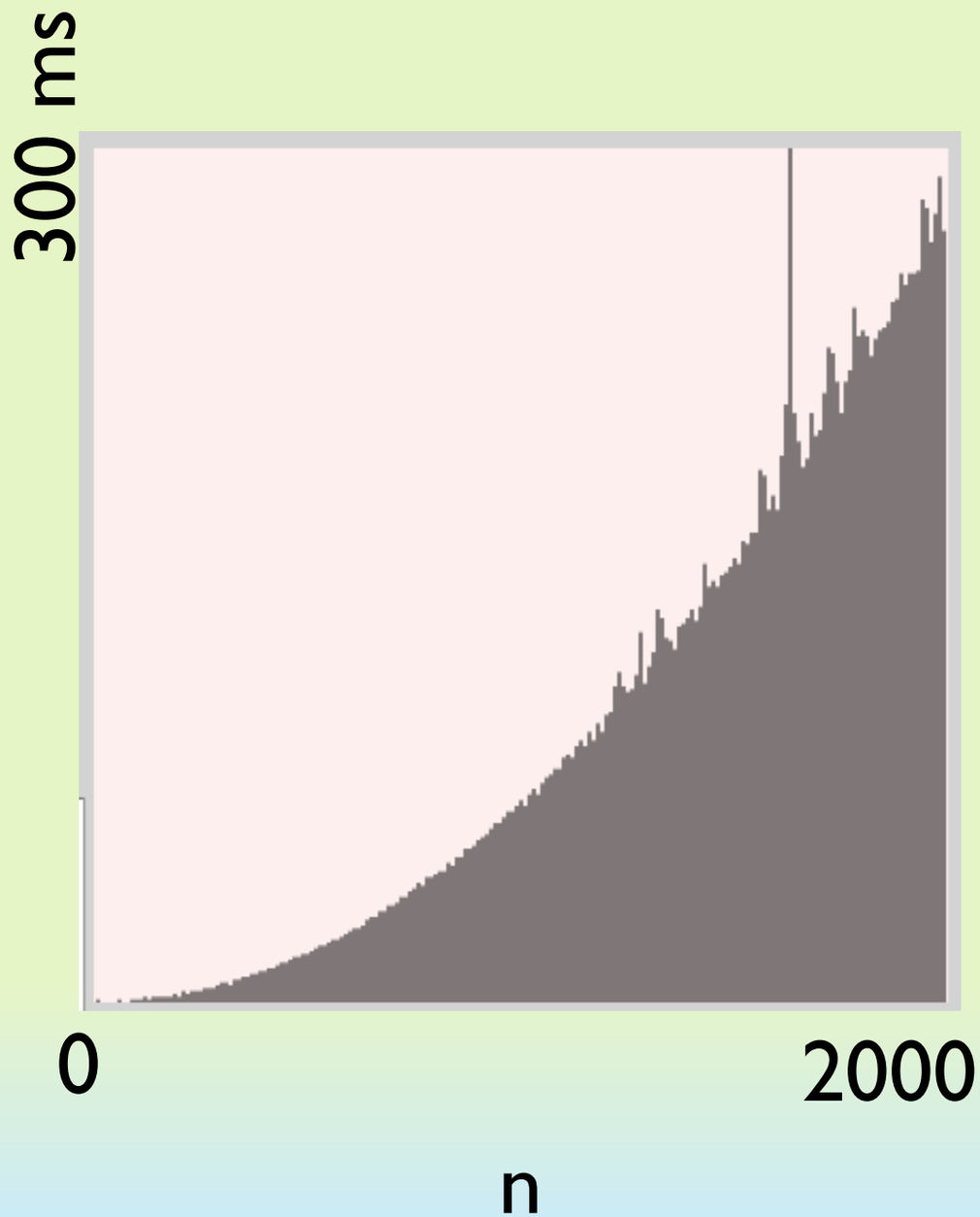
```
function selection_sort(lst){
    var n = lst.length
    for (i = 0; i< n; i++){
      var m = lst[i]
      var mj = i
      for (j = i + 1; j < n; j++){
         if (m > lst[j]){
            m = lst[j]
            mj = j
         }
      }
      var tmp = lst[mj]
      lst[mj] = lst[i]
      lst[i] = tmp
    }
}
```

```
function time_measurement(n){
    var lst = []
    for (var i = 0; i < n; i++){
      lst.push(Math.random())
    }
    var t1 = new Date()
    selection_sort(lst)
    var t2 = new Date()
    return t2 - t1
}
```

# ◇◇ ソーティングに要する計算時間 (3) ◇◇

- 計算時間が２次関数の形になっているのが見てとれる.

```
function measurement(){
    var res = []
    for (var n = 0; n < 2000; n += 10){
      res.push([n, time_measurement(n)])
    }
    for (var i = 0; i < res.length; i++){
      var data = res[i]
      var x = data[0] / 2000
      var y = data[1] / 300
      puts(x + " " + y)
      draw_line([x, 0], [x, y])
    }
}

measurement()
```



# ◇◇ ソーティングに要する計算時間 (4) ◇◇

- これに対して，マージソートの計算量を調べてみる．

```
function merge(lst1, lst2){
  var lst = []
  while (true){
    if (lst1.length == 0) return lst.concat(lst2)
    else if (lst2.length == 0) return lst.concat(lst1)
    if (lst1[0] < lst2[0]) lst.push(lst1.shift())
    else lst.push(lst2.shift())
  }
}

function msort(lst){
  var n = lst.length
  if (n < 2) return lst
  else {
    var n2 = Math.floor(n / 2)
    return merge(
        msort(lst.slice(0, n2)), msort(lst.slice(n2, n)))
  }
}
```

# ◇◇ ソーティングに要する計算時間 (5) ◇◇

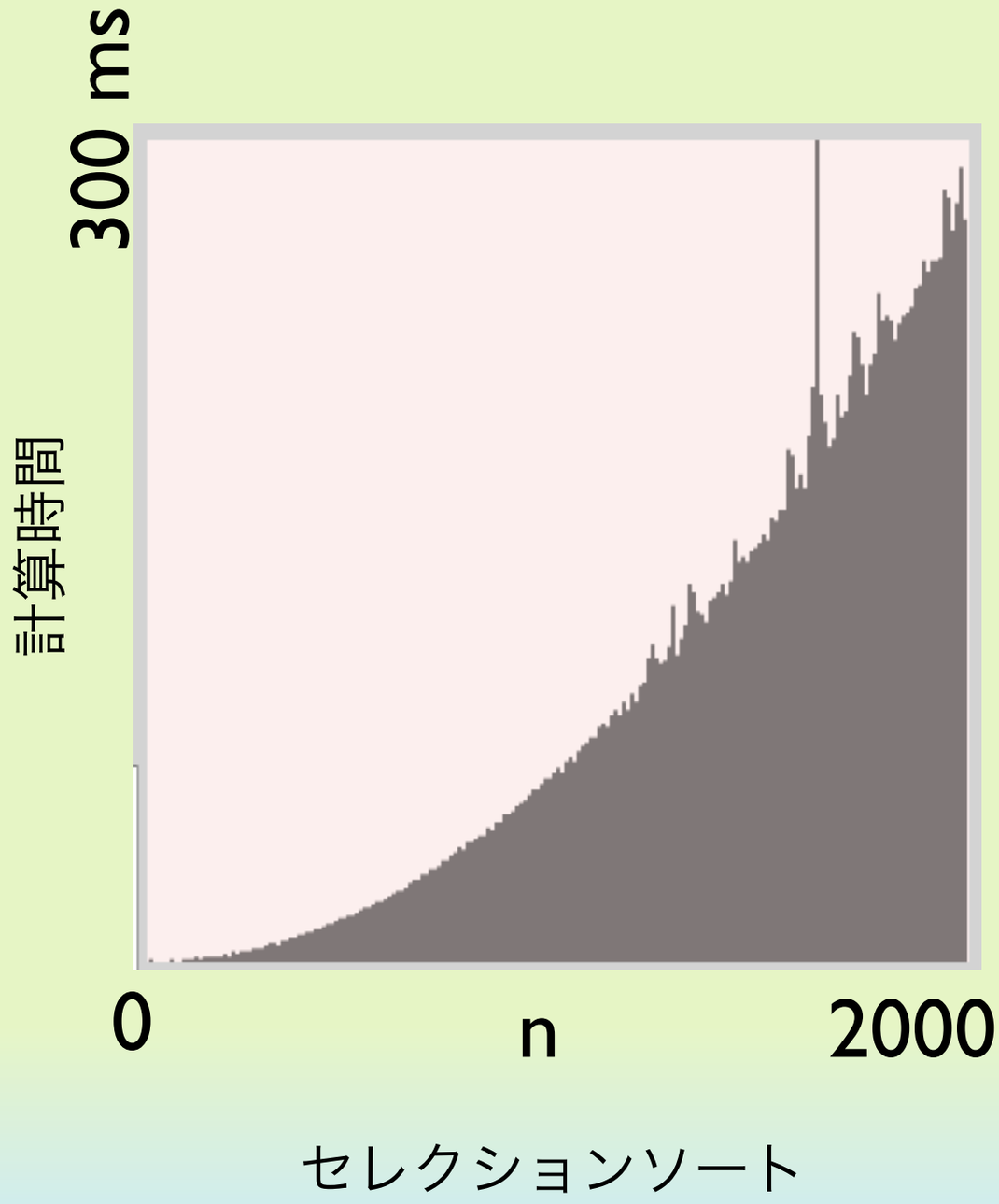
- セレクションソートと同様に時間を測りグラフにする

```
function time_measurement(n){
    var lst = []
    for (var i = 0; i < n; i++){
      lst.push(Math.random())
    }
    var t1 = new Date()
    msort(lst)
    var t2 = new Date()
    return t2 - t1
}
```

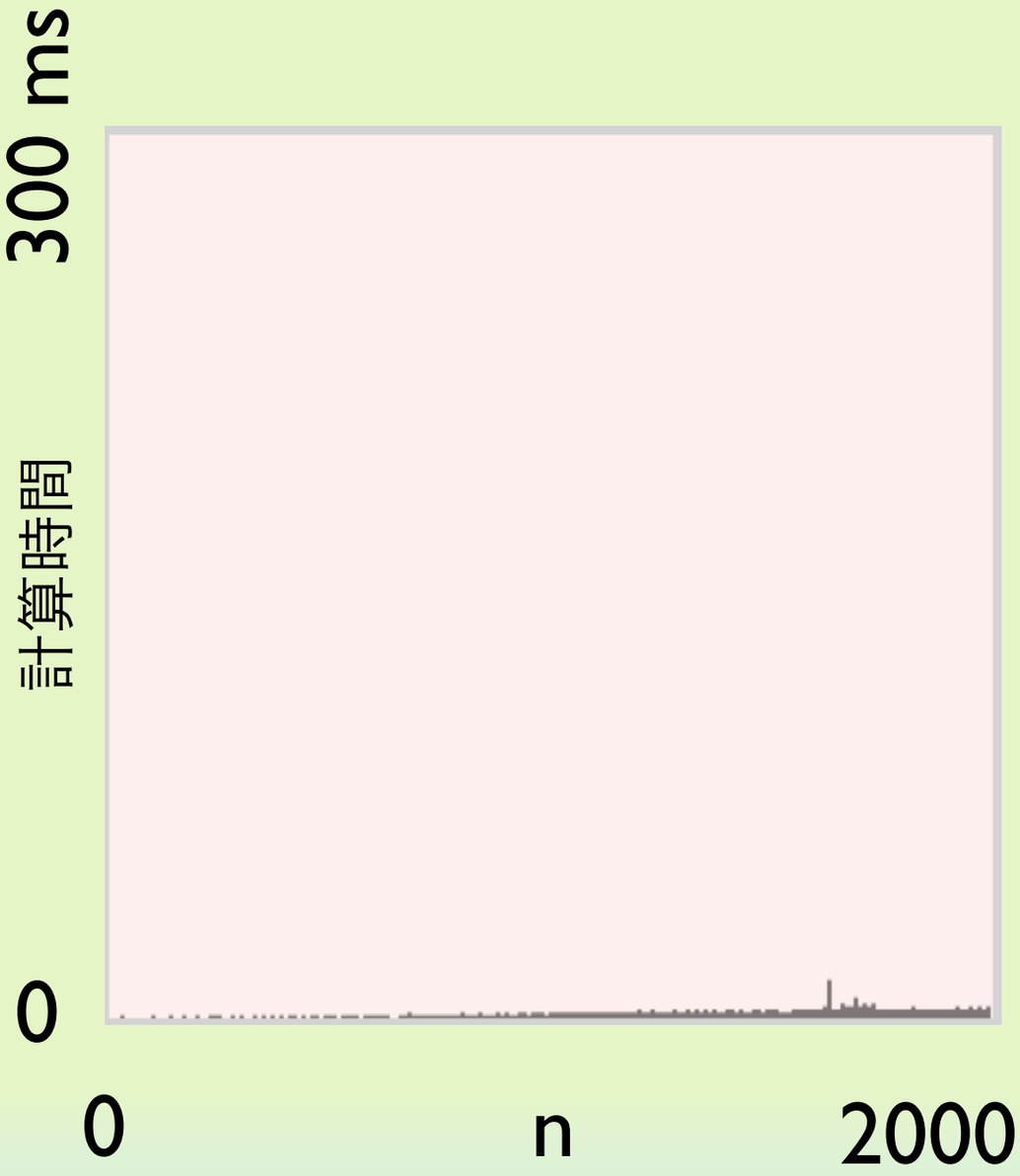
```
function measurement(){
    var res = []
    for (var n = 0; n < 2000; n += 10){
      res.push([n, time_measurement(n)])
    }
    for (var i = 0; i < res.length; i++){
      var data = res[i]
      var x = data[0] / 2000
      var y = data[1] / 300
      puts(x + " " + y)
      draw_line([x, 0], [x, y])
    }
}

measurement()
```

# ◇◇ ソーティングに要する計算時間 (6) ◇◇

右がマージソートの結果である．両者の違いははっきりしている．




セレクションソート




マージソート

# ◇◇ ソーティングに要する計算時間 (7) ◇◇

マージソートの計算量を解析する．n個の要素をソートするに要する時間をT(n)とおく．

↓半分ずつに分けてそれぞれをマージソートする

$$T(n) = 2T(n/2) + cn$$

↑ソートされた2つのソート列をマージする

$S(m) = T(2^m)/2^m$ とおく．このとき，

$$S(m) = (2T(2^m/2) + c2^m)/2^m$$

$$= T(2^{m-1})/2^{m-1} + c = S(m-1) + c$$

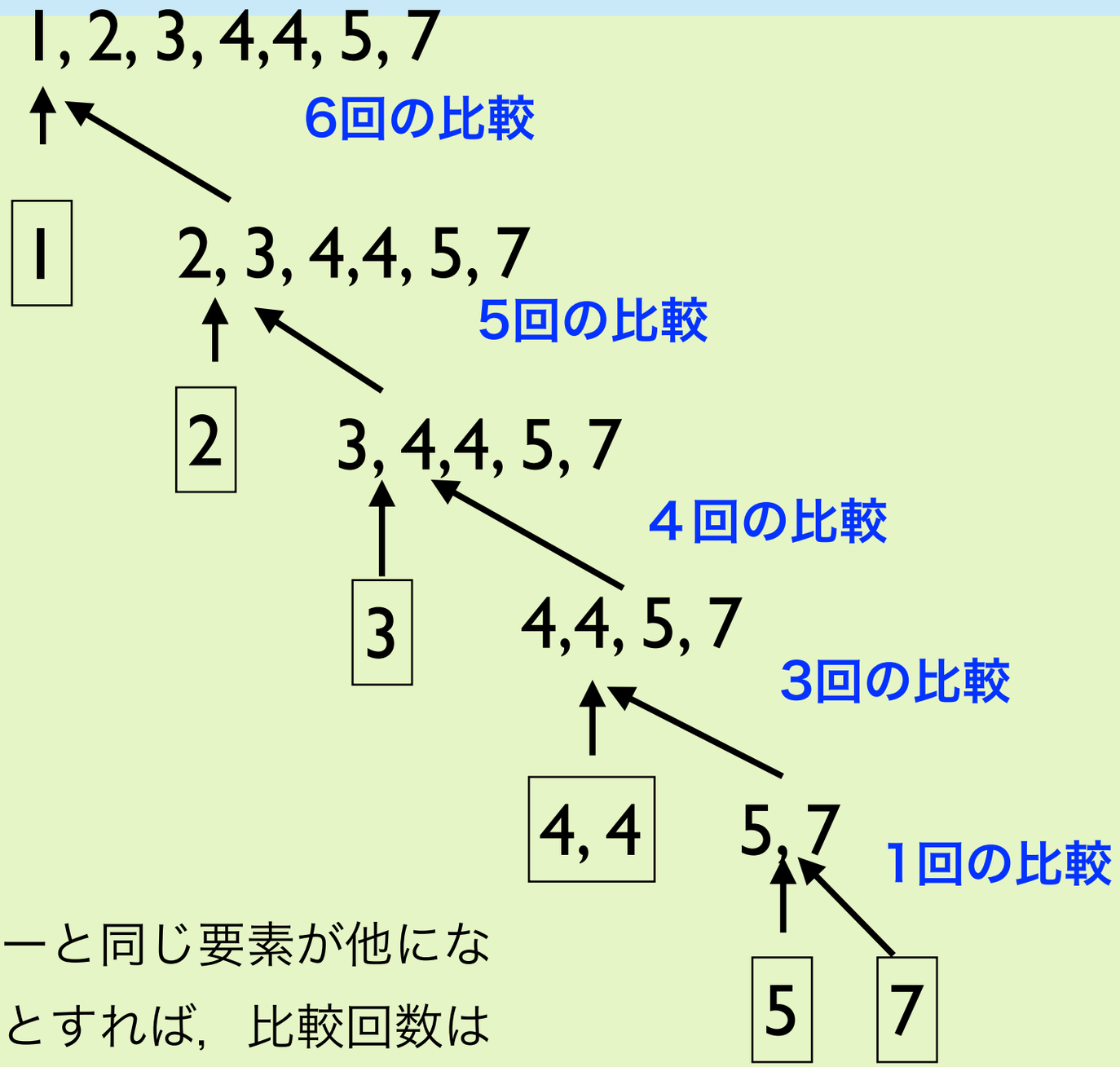
また，$S(0) = T(1) = d$ とすれば，$S(m) = cm + d$

$n = 2^m$ とおけば，

$$T(n) = 2^m S(2^m) = n(cm + d) = cn \log_2 n + nd$$

(c, d は定数)

# クイックソートの計算量 (1)




キーと同じ要素が他にないとすれば，比較回数は

$$T(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$$

セレクションソートと同じ

クイックソートはキーとして何を選ぶかによって，その後の動きが変化する．したがって，データの大小関係によって，計算量が変化する．まずは，もっとも悪い状況下での計算量（最悪計算量）を見積もる．

最も悪い状況では，キーが常に最大値または最小値になっている場合である．

# クイックソートの計算量 (2)

- セレクションソートはランダムなキーが選ばれるから速い.

```
function qsort(lst){
   if (lst.length <= 1) return lst
   var key = lst[0]
   var lt = []
   var eq = []
   var gt = []
   for (var i = 0; i < lst.length; i++){
     var ele = lst[i]
     if (ele < key) lt.push(ele)
     else if (ele > key) gt.push(ele)
     else eq.push(ele)
   }
   return qsort(lt).concat(eq).concat(qsort(gt))
}
var m = []
for (var i = 0; i < 2000; i++)
   m.push(Math.random())
t1 = new Date()
m = qsort(m)
t2 = new Date()
m = qsort(m)
t3 = new Date()
puts("time1 = " + (t2 - t1) + " ms;  " +
      "time2 = " + (t3 - t2) + " ms")
```

１回目のqsortはランダムな列のソートなので速いが，２回目のqsortは既にソートされた列なので非常に時間がかかる.

time1 = 4 ms;  time2 = 150 ms

# ◇◇ クイックソートの計算量 (3) ◇◇

どの要素がキーとして選ばれる確率も等しいと仮定したときの平均時間計算量を計算してみる．E(n)を長さnの列をソートしたときの平均の時間とする．すると，

$$E(n) = cn + \frac{1}{n}\sum_{j=1}^{n}(E(j-1) + E(n-j))$$

nを両辺にかけて，nとn - 1を代入すると，以下の様になる

$$nE(n) = cn^2 + \sum_{j=1}^{n}(E(j-1) + E(n-j))$$

$$(n-1)E(n-1) = c(n-1)^2 + \sum_{j=1}^{n-1}(E(j-1) + E(n-1-j))$$

この２式の引き算を計算すれば，

$$nE(n) - (n-1)E(n-1) = c(2n-1) + 2E(n-1)$$

# クイックソートの計算量 (4)

結局以下のように変形される

$$nE(n) - (n+1)E(n-1) = c(2n-1)$$

さらに，両辺をn(n + 1)で割ると

$$\frac{E(n)}{n+1} - \frac{E(n-1)}{n} = c\frac{2n-1}{n(n+1)}$$

となる．E(0) = 0とおけば,

$$\frac{E(n)}{n+1} - \frac{E(0)}{1} = \frac{E(n)}{n+1} = c\sum_{i=1}^{n}\frac{2i-1}{i(i+1)}$$

これより,

$$E(n) = c(n+1)\sum_{i=1}^{n}\frac{2i-1}{i(i+1)} = c(n+1)\sum_{i=1}^{n}\left(\frac{3}{i+1} - \frac{1}{i}\right)$$

# クイックソートの計算量 (5)

ここで,

$$\sum_{i=1}^{n}\left(\frac{3}{i+1}-\frac{1}{i}\right)=\sum_{i=2}^{n+1}\frac{3}{i}-\sum_{i=1}^{n}\frac{1}{i}=\frac{3}{n+1}-3+\sum_{i=1}^{n}\frac{2}{i}$$

また，nがある程度大きいときは以下の近似式が成り立つ

$$1+\frac{1}{2}+\cdots+\frac{1}{n}\approx\log_e n$$

これより,

$$E(n)=c(n+1)\left(2\log_e n-3\frac{n}{n+1}\right)$$

$$=2c(n+1)\log_e n-3cn$$

**log n を定数と考えれば，実行時間はほぼ線形に近い**

# クイックソートの計算量 (6)

```
function qsort(lst){
   n = lst.length
   if (n <= 1) return lst
   var key = lst[Math.floor(Math.random() * n)]
   var lt = []
   var eq = []
   var gt = []
   for (var i = 0; i < lst.length; i++){
     var ele = lst[i]
     if (ele < key) lt.push(ele)
     else if (ele > key) gt.push(ele)
     else eq.push(ele)
   }
   return qsort(lt).concat(eq).concat(qsort(gt))
}

var m = []
for (var i = 0; i < 2000; i++)
   m.push(Math.random())
t1 = new Date()
m = qsort(m)
t2 = new Date()
m = qsort(m)
t3 = new Date()
puts("time1 = " + (t2 - t1) + " ms;  " +
      "time2 = " + (t3 - t2) + " ms")
```

キーを選択するときランダムに選択すると，実用上ほぼ高速に動作するクイックソートアルゴリズムを作ることができる．

time1 = 5 ms;  time2 = 3 ms

# ◇◇ ソーティングアルゴリズムの計算量 ◇◇

- それぞれのソートアルゴリズムは計算量において違いがある

| アルゴリズム | 最悪実行時間 | 平均実行時間 | 空間計算量 |
|---|---|---|---|
| セレクションソート | n(n - 1)/2 | n(n - 1)/2 | 定数 |
| バブルソート | n(n - 1)/2 | n(n - 1)/2 | 定数 |
| クイックソート | n(n - 1)/2 | c (n + 1) log n | n (実はlogn) |
| ヒープソート | n log n | n log n | log n |
| マージソート | n log n | n log n | n |