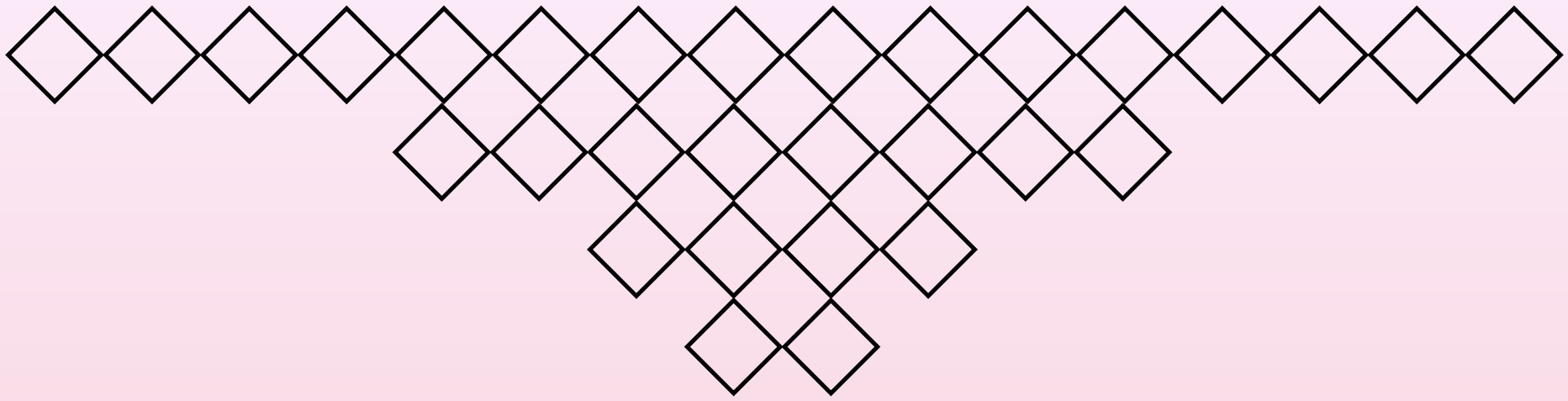
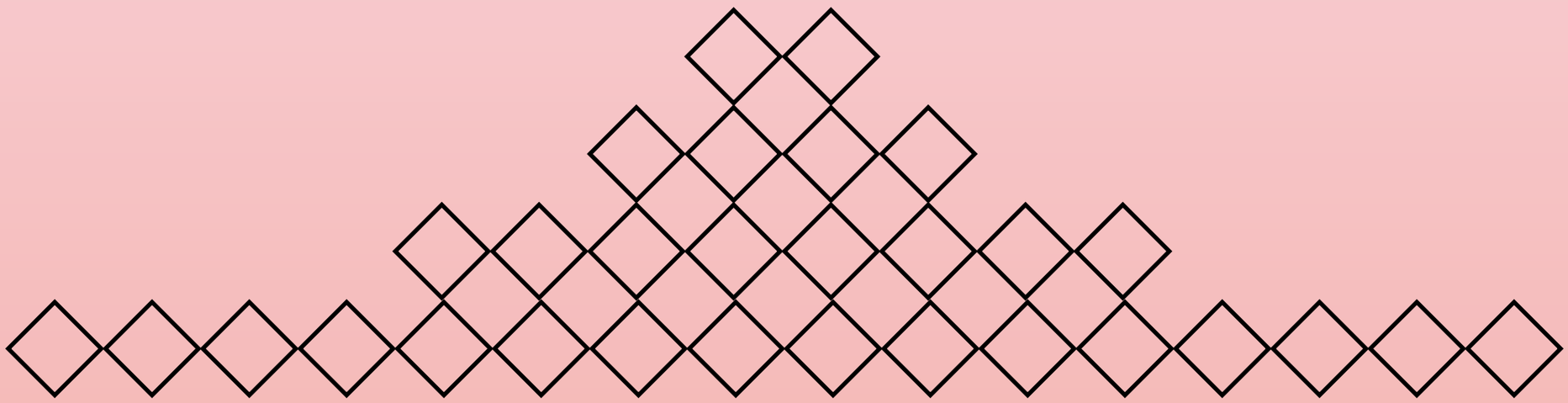


アルゴリズム・データ構造 I 第8回
順列と組み合せの生成

名城大学理工学部情報工学科

山本修身



与えられた木のノードを順に走査する方法：深さ優先探索 (1)

復習

木構造を陽に作って計算するか否かは別にして，木構造を頭の中に描いてアルゴリズムを設計することは非常に多い．その際，木に対して必要な操作として，ノードを順に走査 (scan) することが重要になる．ノードをどの順に走査するかは任意性があり，よく使われる方法として，**深さ優先探索（depth-first search)** と**幅優先探索 (breadth-first search)** がある．ここではプログラミングが簡単な深さ優先探索から見て行く．

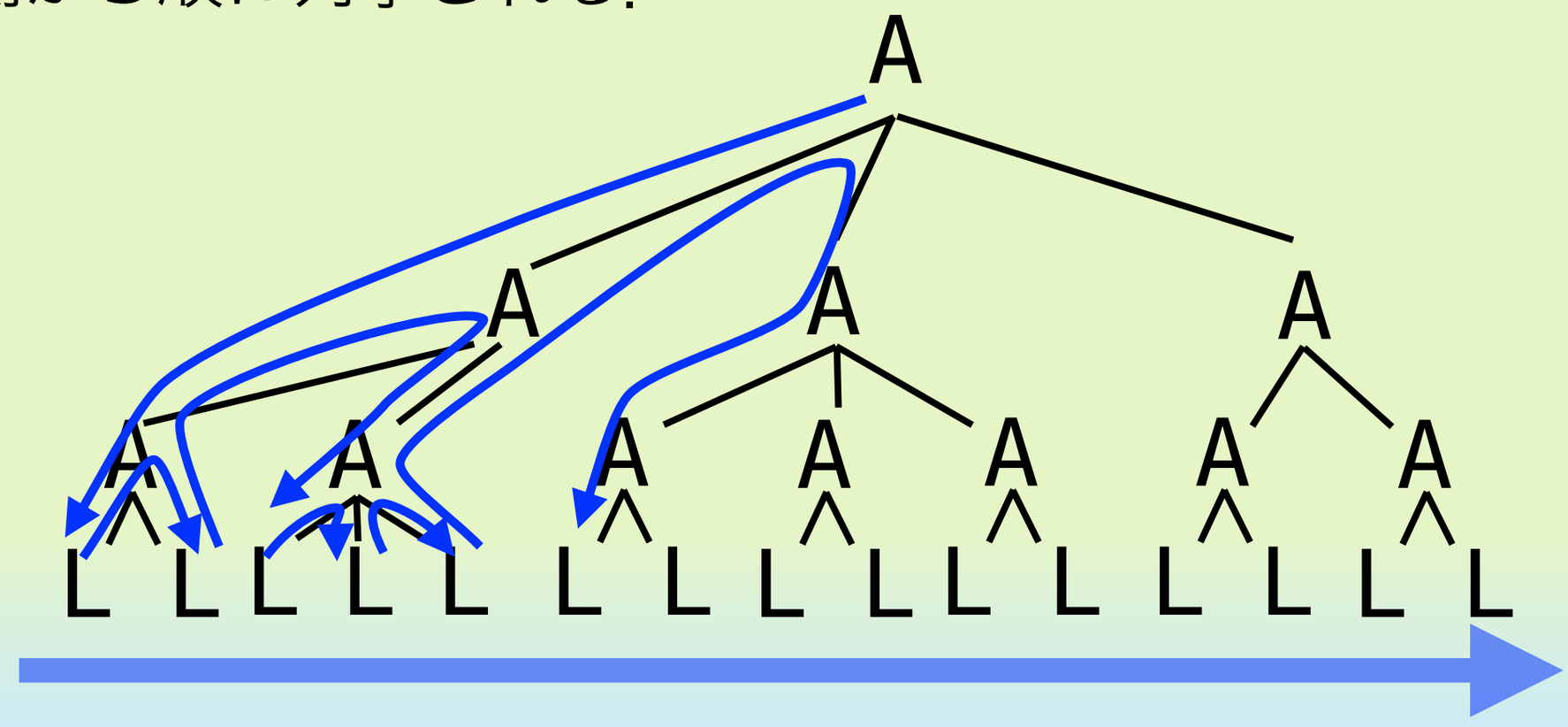
与えられた木のノードを順に走査する方法：深さ優先探索 (2)

復習

深さ優先探索は葉に出会うまで，深く進み，深く進めなくなったら，一段階もどって，別の方向に深く進む．深く進むことを優先する探索法である．

もし，木が無限に深い場合には止まらなくなる．したがって，この方法は使えない．

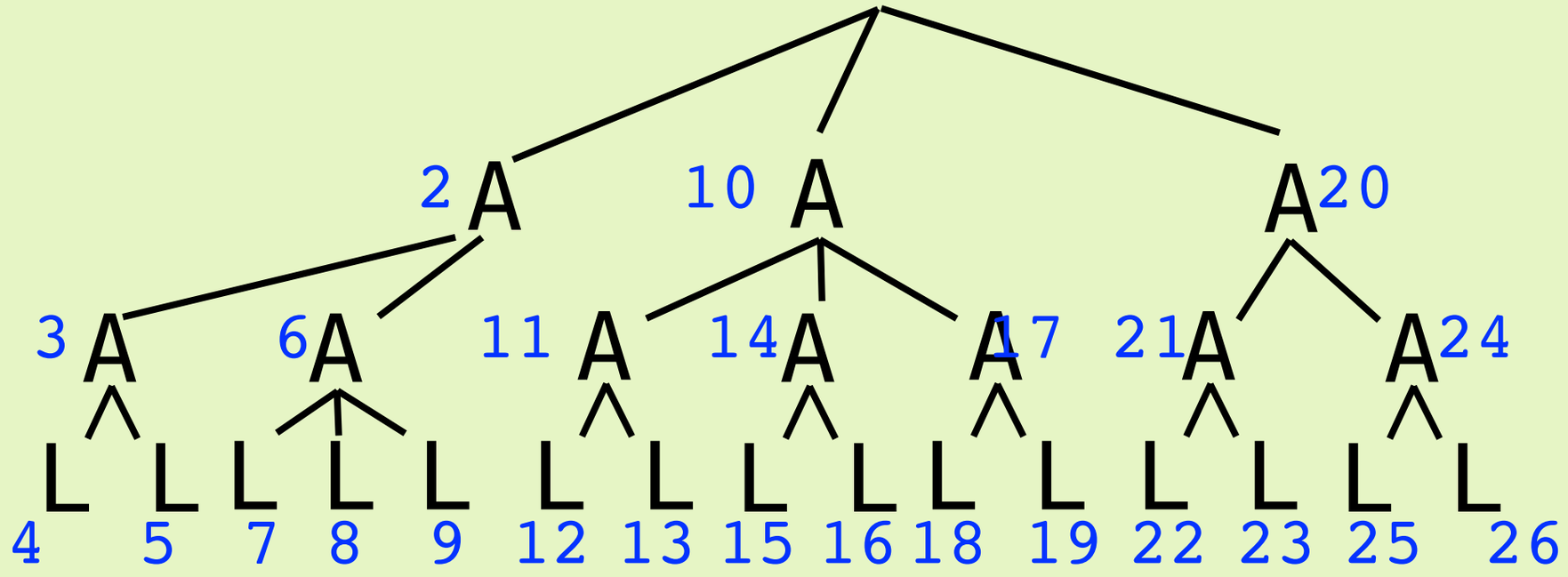
葉は左側から順に列挙される．



与えられた木のノードを順に走査する方法：深さ優先探索 (3)

復習

葉ではないノードの順位も含めて以下のような順番で探索が行われる.



与えられた木のノードを順に走査する方法：深さ優先探索 (4)

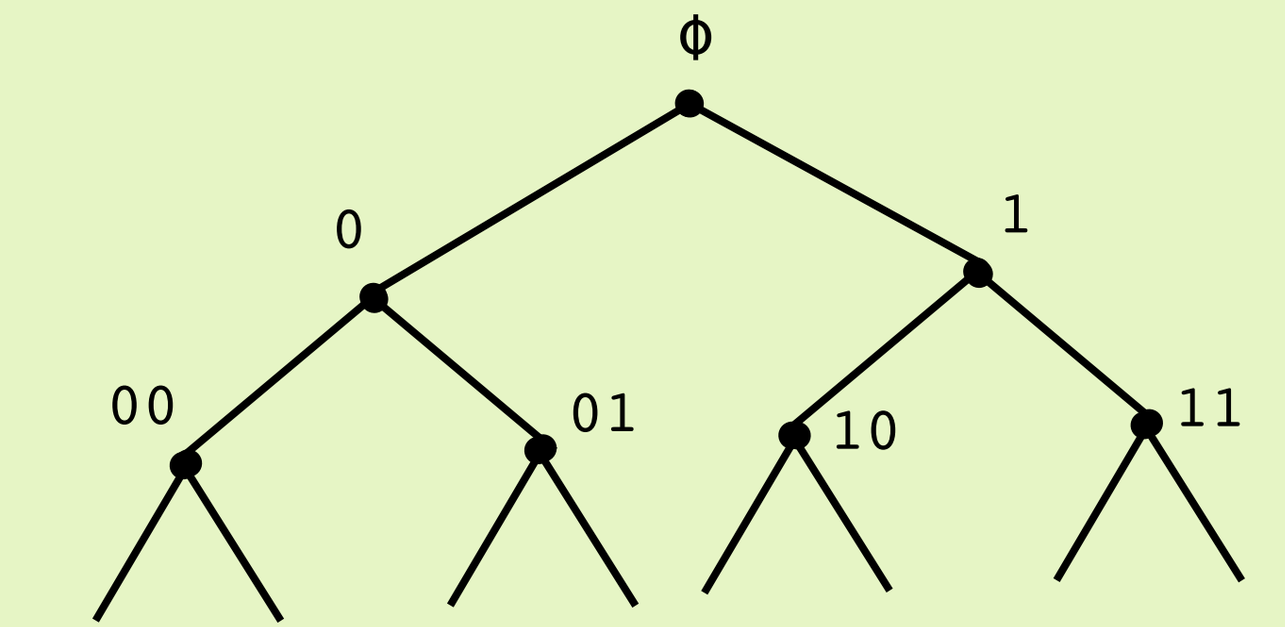
復習

深さ優先探索を行うためのプログラム

```
function DFS(node){
    nodeについての処理;
    for (nodeのすべての子供childについて) {
       DFS(child)
    }
}
```

◇◇深さ優先探索の例：すべての2進数を表示する◇◇

n桁の2進数をすべて表示する.



出力

```
0000000000
0000000001
0000000010
..........
..........
1111111101
1111111110
1111111111
```

```
var b = []
var n = 10
function print_bin(){
  var s = ""
  for (var i = 0; i < n; i++){
    s += b[i]
  }
  puts(s)
}
```

nビットを表示する

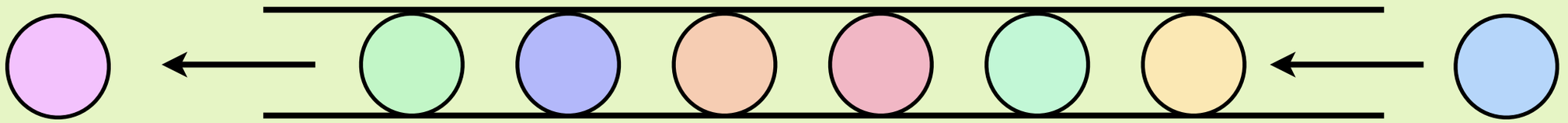
k番目のビットを決定する

```
function binary(k){
  if (k >= n){
    print_bin();
  } else {
    b[k] = 0
    binary(k + 1)
    b[k] = 1
    binary(k + 1)
  }
}

binary(0)
```

◇◇もう一つの探索方法：幅優先探索 (1)◇◇

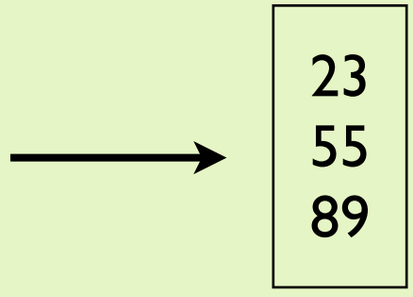
データ構造：キュー（queue; FIFO: **F**irst **I**n **F**irst **O**ut)



取り出すときは左から

入れるときは右から

```
a = []
a.push(23)
a.push(55)
puts(a.shift())
a.push(89)
puts(a.shift())
puts(a.shift())
```



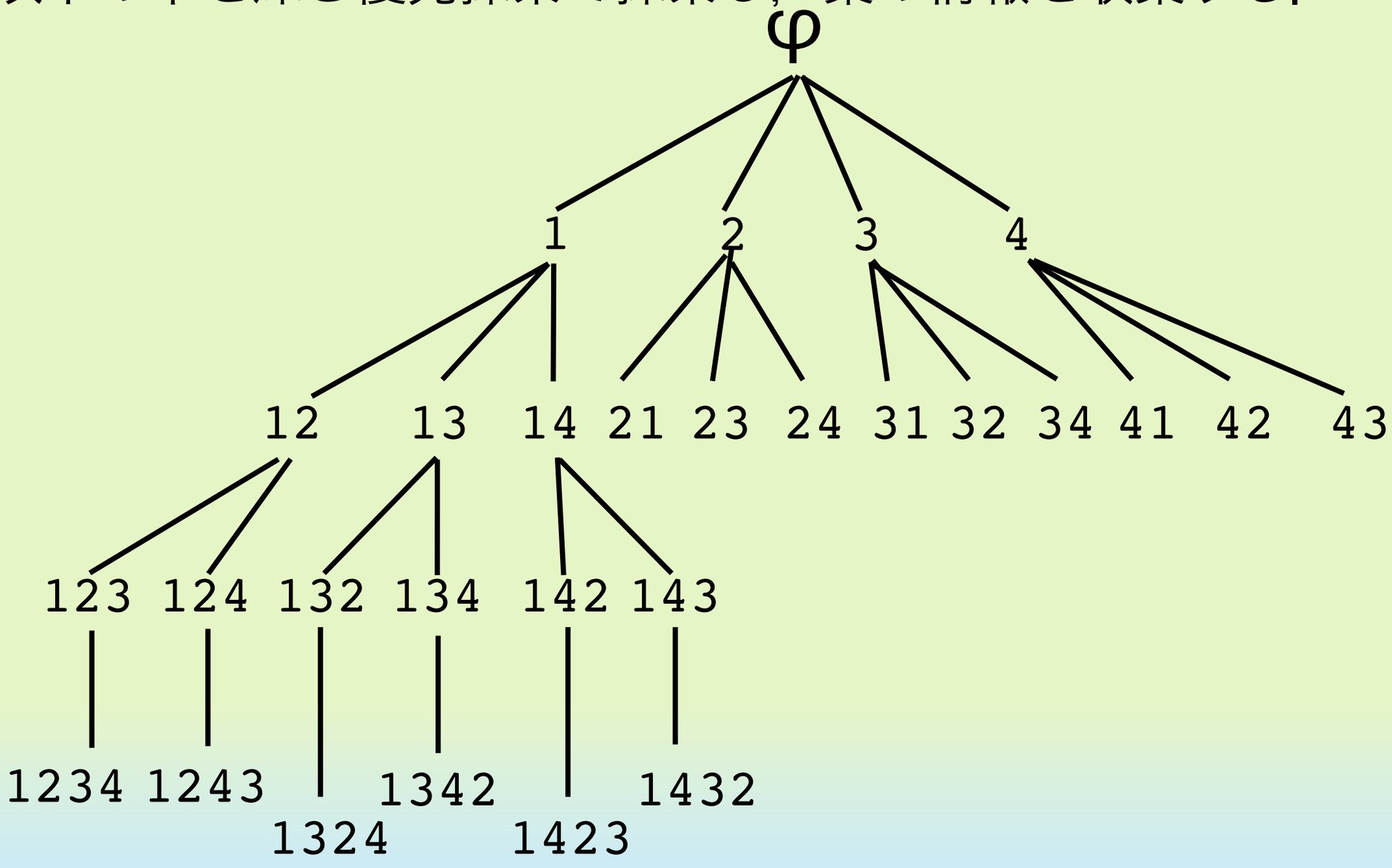
JavaScriptでは，配列にキューの機能がついている．データを入れる場合にはpush(), データを取り出す場合にshift()を用いる

順列の生成 (1)

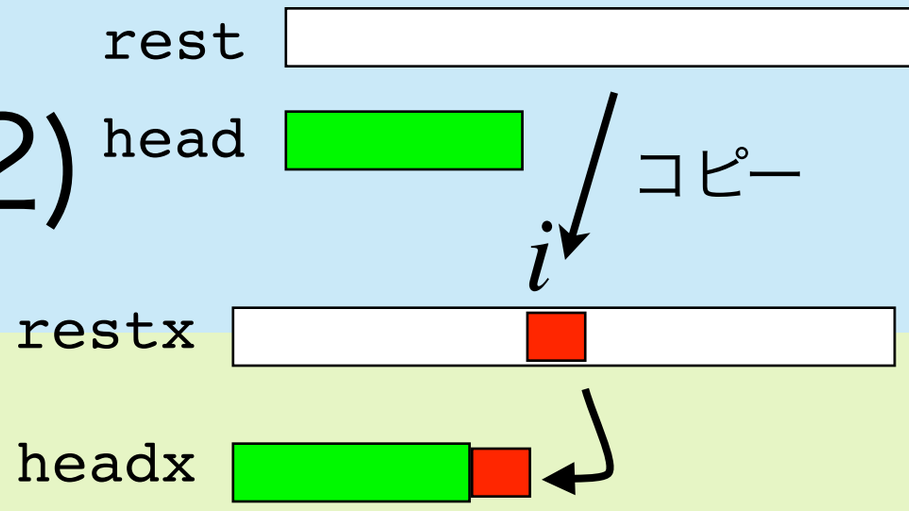
組み合せ問題を解くために順列の生成は不可欠である.

まずは, 単純な順列の生成方法について考える.

以下の木を深さ優先探索で探索し, 葉の情報を収集する.



順列の生成 (2)



順列を深さ優先探索で生成する

```
function perm(head, rest){
   if (rest.length == 0) return [head]
   else {
     var res = []
     for (var i = 0; i < rest.length; i++){
        var restx = rest.slice(0)
        var headx = head.concat(restx.splice(i, 1))
        res = res.concat(perm(headx, restx))
     }
     return res
   }
}

m = perm([], [1, 2, 3, 4])
puts(m.length)
for (var i = 0; i < m.length; i++)
   puts(m[i].join("-"))
```

←restをrestxへコピー

restxのi番目をheadの後ろに追加してheadxにする

restxのi番目は削除

```
24
1-2-3-4
1-2-4-3
1-3-2-4
1-3-4-2
1-4-2-3
1-4-3-2
2-1-3-4
2-1-4-3
2-3-1-4
2-3-4-1
2-4-1-3
2-4-3-1
3-1-2-4
3-1-4-2
3-2-1-4
3-2-4-1
3-4-1-2
3-4-2-1
4-1-2-3
4-1-3-2
4-2-1-3
4-2-3-1
4-3-1-2
4-3-2-1
```

順列の生成 (3)

配列のいつかのメソッドの利用方法

slice(0) 【配列のコピー】

```
js> a = [3, 4, 5]
[3, 4, 5]
js> b = a.slice(0)
[3, 4, 5]
js> b[0] = 100
100
js> a
[3, 4, 5]
js> b
[100, 4, 5]
```

splice(i, 1) 【配列一部を削除】

```
js> a = [3, 4, 5]
[3, 4, 5]
js> a.splice(1, 1)
[4]
js> a
[3, 5]
```

concat(b) 【配列の繋ぎ合わせ】

```
js> a = [3, 4, 5]
[3, 4, 5]
js> b = a.concat([20, 30])
[3, 4, 5, 20, 30]
js> b
[3, 4, 5, 20, 30]
js> a
[3, 4, 5]
```

join(“---”) 【配列要素の間に文字列を挟んで文字列を作る】

```
js> a = [3, 4, 5]
[3, 4, 5]
js> a.join(":::")
"3:::4:::5"
```

順列の生成 (4)

permの使い方

- perm(初期の順列［普通は空列］, これから利用する記号列)

```
m = perm(['d'], ['a', 'b', 'c'])
puts(m.length)
for (var i = 0; i < m.length; i++)
   puts(m[i].join("-"))
```

↓

```
6
d-a-b-c
d-a-c-b
d-b-a-c
d-b-c-a
d-c-a-b
d-c-b-a
```

魔方陣を作ろう (1)

ここでは，最も簡単な3x3の魔方陣を計算してみる.

魔方陣とは，nxnのマスに1〜n^2の数を入れて縦，横および斜めの数の和がすべて等しい配置にしたものである.

左の配置は魔方陣となっている．この場合，縦横斜めそれぞれの和は15となっている．これは以下の式によって計算することができる.

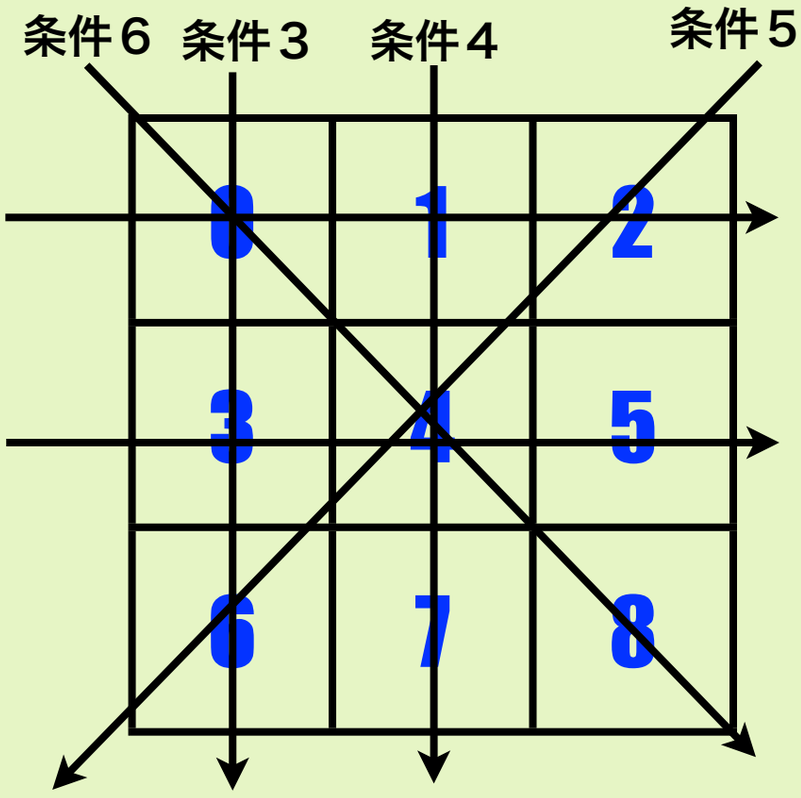
2	7	6
9	5	1
4	3	8

$$\frac{1+2+\cdots+9}{3}=15$$ 魔法和

魔方陣を作ろう (2)

ここでは，前述の順列で3x3の領域での並べ方を全て列挙して，その中で魔方陣になっている順列だけを表示させる.

魔方陣の条件としては，右に示す6つの和が15になるという条件がすべて満たされれば良い.




```
a = []                    ←魔方陣に使う要素の作成
for (i = 1; i < 10; i++) a[i - 1] = i
m = perm([], a)                ←順列の計算
puts(m.length)
for (var i = 0; i < m.length; i++){
    mm = m[i]
    if (mm[0] + mm[1] + mm[2] != 15) continue
    if (mm[3] + mm[4] + mm[5] != 15) continue
    if (mm[0] + mm[3] + mm[6] != 15) continue
    if (mm[1] + mm[4] + mm[7] != 15) continue
    if (mm[2] + mm[4] + mm[6] != 15) continue
    if (mm[0] + mm[4] + mm[8] != 15) continue
    puts(mm.join("-"))
}
```

魔方陣を作ろう (3)

前のスライドのプログラムを実行すると以下のように表示される.

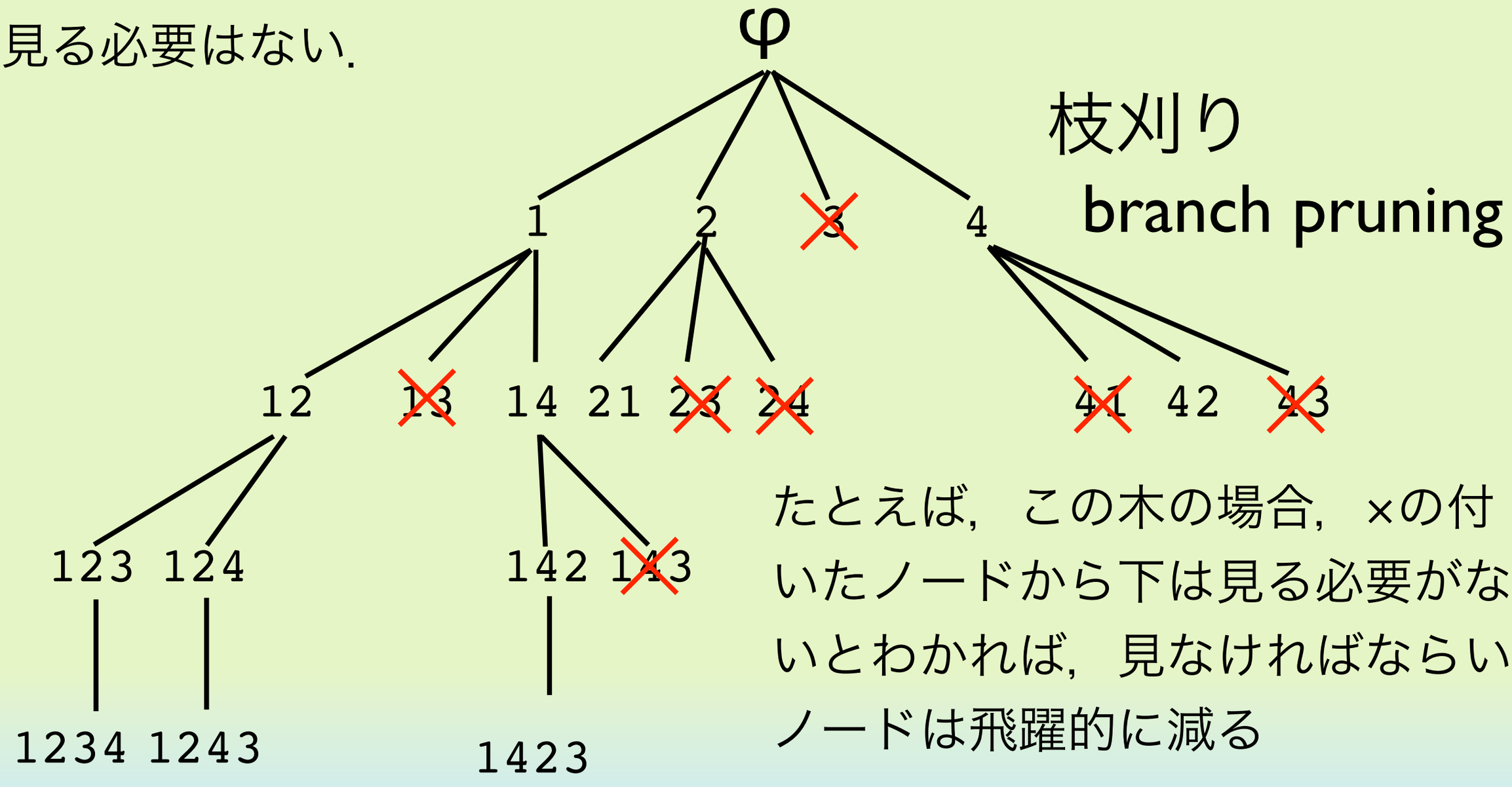
```
362880   ←最初の順列の総数（すなわち9!）
2-7-6-9-5-1-4-3-8
2-9-4-7-5-3-6-1-8
4-3-8-9-5-1-2-7-6
4-9-2-3-5-7-8-1-6
6-1-8-7-5-3-2-9-4
6-7-2-1-5-9-8-3-4
8-1-6-3-5-7-4-9-2
8-3-4-1-5-9-6-7-2
```

←条件をクリアーした数の配置

実はこの配置は本質的にすべて同じ，回転による4つの配置とそれぞれを裏返した4つの配置に他ならない.

魔方陣を作ろう (4)

前述のプログラムは，一度 9! 個の順列を発生させてから条件に合う順列だけを取り出すという効率の悪い方法を採用している．あらかじめ最初の段階で，駄目だとわかれば，そのノードよりも下のノードを見る必要はない．

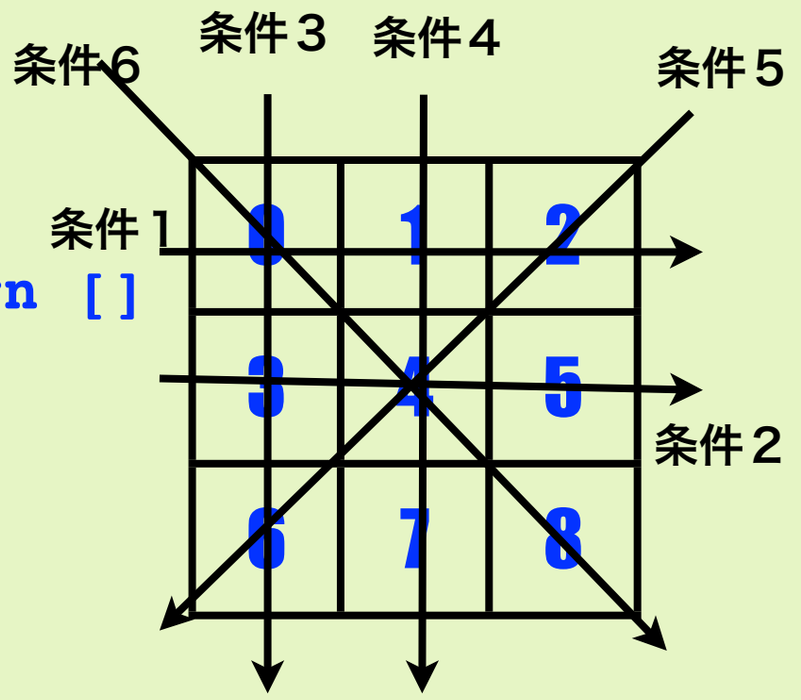


たとえば，この木の場合，×の付いたノードから下は見る必要がないとわかれば，見なければならいノードは飛躍的に減る

魔方陣を作ろう (5)

青色の部分を追加することにより枝刈りできる.

```
function perm(head, rest){
   if (head.length == 3 &&
         head[0] + head[1] + head[2] != 15) return []
   if (rest.length == 0) return [head]
   else {
     var res = []
     for (var i = 0; i < rest.length; i++){
        var restx = rest.slice(0)
        var headx = head.concat(restx.splice(i, 1))
        res = res.concat(perm(headx, restx))
     }
     return res
   }
}
```



答は変わらないが，得られる順列の個数が飛躍的に少なくなっている．もとは362880だった.

出力結果

```
34560
2-7-6-9-5-1-4-3-8
2-9-4-7-5-3-6-1-8
4-3-8-9-5-1-2-7-6
4-9-2-3-5-7-8-1-6
6-1-8-7-5-3-2-9-4
6-7-2-1-5-9-8-3-4
8-1-6-3-5-7-4-9-2
8-3-4-1-5-9-6-7-2
```

ナップザック問題 (1)

ランダムに発生させた20個の乱数（0〜999の数）の中から適当にいくつかの数を選択して，その和が1000を超えないで，1000に最も近くなるような組み合せを見つけるプログラムを考える.

たとえば,

342, 526, 773, 756, 994, 875, 674, 287, 832,
674, 940, **493**, 10, 717, **165**, 26, 523, 611,
760, 546

の中から適当に選んでなるべく1000に近くしたい．この場合,

342 + 493 + 165 = 1000

とすることができる.

この問題は一般的には**NP-完全**とよばれるクラスに属する問題で，非常に解くのが難しいことが知られている．少ない個数の問題については実際には解くことができる.

ナップザック問題 (2)

knapsackという関数を考える.

`knapsack( 用いる数, 上限 ) = 選択された数のリスト`

```
knapsack([43, 22, 44, 55], 100) = [43,44]
```

```
knapsack([a1,a2,...,an], limit)
```

limitに近い方を選択

```
knapsack([a2,...,an], limit)
```

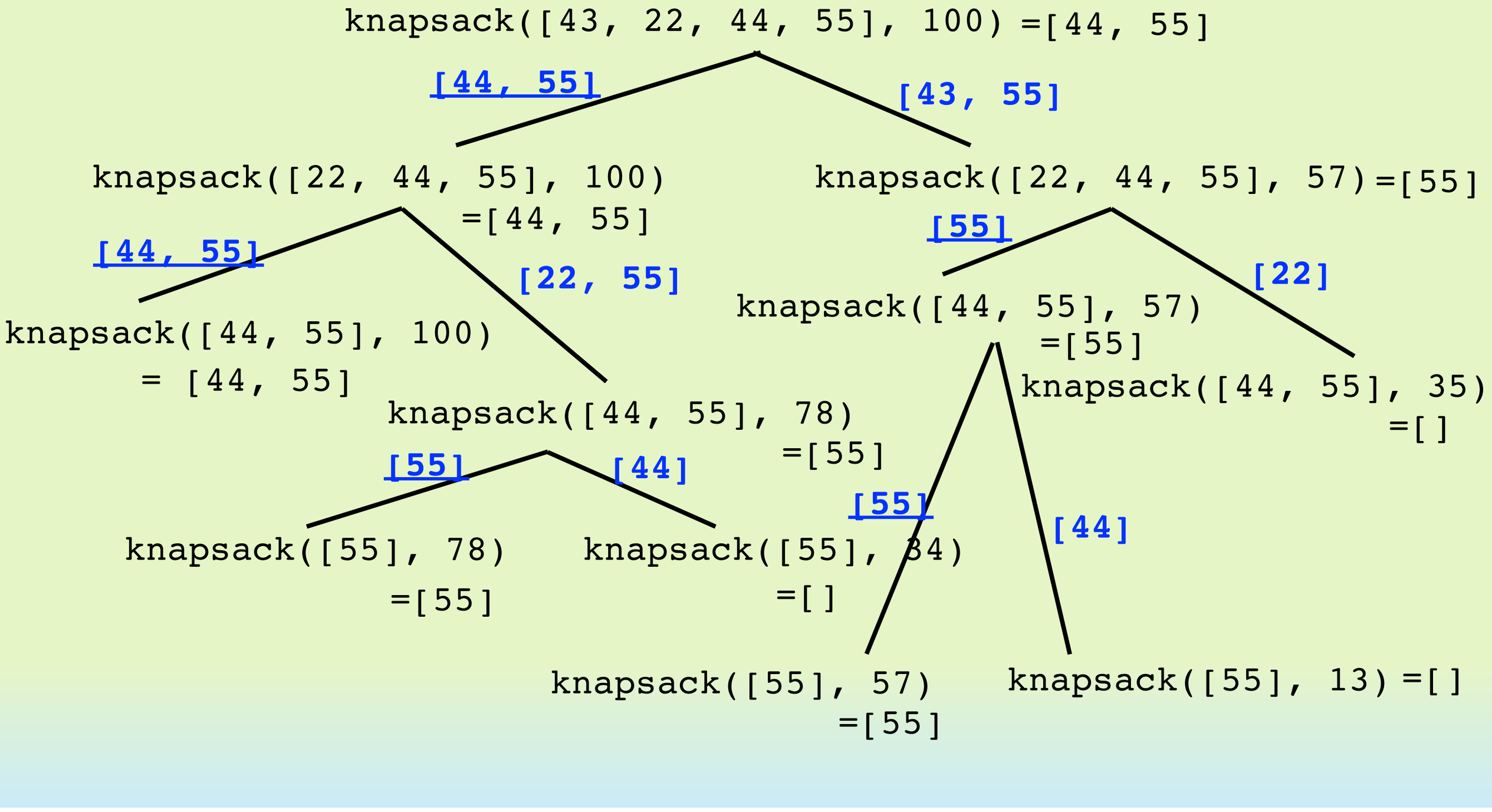
a1を使わない場合

```
knapsack([a2,...,an], limit-a1)
```

a1を使う場合

ナップザック問題 (3)

knapsack関数の計算例．木を辿ることになる．深さ優先探索で計算



ナップザック問題 (4)

以下のような関数をまず用意する．filterはlstの要素でlimitを超えないものだけを取り出して返す関数である．また，sumはlstの要素の和を計算する．

```
function filter(lst, limit){
   var lst1 = []
   for (var i = 0; i < lst.length; i++)
     if (lst[i] <= limit) lst1.push(lst[i])
   return lst1
}

function sum(lst){
   var s = 0
   for (var i = 0; i < lst.length; i++)
     s = s + lst[i]
   return s
}
```

ナップザック問題 (5)

プログラムは以下のとおり

```
function knapsack(lst, limit){
  var llst = filter(lst, limit)
  if (llst.length == 0) return []
  else if (sum(llst) < limit) return llst
  else {
    var a = llst.shift()
    var m1 = knapsack(llst, limit)
    var m2 = knapsack(llst, limit - a)
    m2.unshift(a)
    if (sum(m1) < sum(m2)) return m2
    else return m1
  }
}
```

←limitよりも大きい要素を削除

←先頭の数を取得

←m2の先頭にaを追加

←大きい方を返す

```
173, 794, 76, 891, 994,
492, 931, 11, 445, 549,
173, 429, 414, 816,
951, 232, 936, 0, 805,
683

11 + 173 + 816 = 1000
```

```
var a = []
for (var i = 0; i < 20; i++)
   a.push(Math.floor(Math.random() * 1000))
puts(a)
ans = knapsack(a, 1000)
puts(ans.join(" + ") + " = " + sum(ans))
```