プログラミング応用 第5回

河瀬 康志

2018年7月9日

アウトライン

演習問題 (1/2)

問 1

次の関数の計算量のオーダーを求めよ．また，$n = 1000, 2000, \ldots, 10000$ の場合について実行時間を計測し，matplotlib を用いてプロットせよ．さらに，関数 ax^b を用いて，最小二乗法により計測結果を近似した結果もプロットせよ．

```
def calc(n):
    res = 0
    for i in range(n):
        for j in range(i):
            res+=j
    return res
```

問 2

次の迷路について，スタートから到達可能なマスの数と，スタートからゴールまでの最短距離を求めよ．

http://yambi.jp/lecture/advanced_programming2018/maze3.txt

演習問題 (2/2)

問 3 (おまけ)

大きさが $h \times w$ マスの庭がある．そこに雨が降り，水たまりができたとするとき，全部でいくつの水たまりがあるか数えたい．ただし，水たまりは 8 近傍で隣接している場合につながっているとみなす．

入力例

```
W........WW.
.WWW.....WWW
.W..WW...WW.
..W......WW.
........WW..
..W......W..
.W.W......W.
W.W.WW...W.W
..WW........
```

W は水たまりを表すとする．
この例の答えは 3 個である．

次の入力について数えよ．

http://yambi.jp/lecture/advanced_programming2018/lake.txt

授業スケジュール

	日程	内容
第 1 回	6/11	ガイダンス・復習
第 2 回	6/18	文字列操作（文字列整形，パターンマッチ，正規表現） 平面幾何（線分の交差判定，点と直線の距離，凸包）
第 3 回	6/25	乱数（一様分布，正規分布への変換，乱数生成） 統計（データ処理，フィッティング）
第 4 回	7/2	計算量（オーダー表記） スタックとキュー（幅優先探索，深さ優先探索）
第 5 回	7/9	ソートアルゴリズム バックトラック（N クイーン問題，数独）
第 6 回	7/23	動的計画法（ナップサック問題） 最短経路探索（Warshall-Floyd, Bellman-Ford, Dijkstra）
第 7 回	7/30	巡回セールスマン問題
期末試験	8/6	南 4 号館 3 階 第 1 演習室で実施

アウトライン

ソートアルゴリズム

- 与えられた数の列を小さい順に並べ替える
  - ソート前：[8, 4, 5, 6, 1, 2, 7]，ソート後：[1, 2, 4, 5, 6, 7, 8]
- 目標
  - sorted をメソッド使わずにソートをできるようになる
  - 効率的にソートをするためにはどのようにすればよいかを学ぶ
  - 参考：`https://www.toptal.com/developers/sorting-algorithms`

```
>>> sorted([8,4,5,6,1,2,7])
[1, 2, 4, 5, 6, 7, 8]
```

バブルソート

最も単純な方法の１つ

- 隣合う要素を比較し，順序が逆になっていれば入れ替える
- 一度リストを捜査すると，最大の要素が最後に移動する
- 計算量は $O(n^2)$ (n はリストの長さ)

```
def bubble_sort(l):
    for i in range(len(l)):
        for j in range(len(l)-i-1):
            if l[j]>l[j+1]:
                l[j],l[j+1]=l[j+1],l[j]
```

マージソート

- 分割統治法を用いてソートする
  - 前半分と後ろ半分をそれぞれソート (サイズ $n/2$ の問題 2 つ)
  - 2 つの結果をくっつける ($O(n)$ 時間)
- 計算量は $O(n \log n)$

```
def merge_sort(l):
    _merge_sort(l,0,len(l))
def _merge_sort(l,s,t):
    if t-s<=1: return
    m=(s+t)//2
    _merge_sort(l,s,m) # 前半をソート
    _merge_sort(l,m,t) # 後半をソート
    a,j,k = [],s,m
    # merge
    for i in range(s,t):
        if k==t or (j<m and l[j]<l[k]):
            a.append(l[j])
            j+=1
        else:
            a.append(l[k])
            k+=1
    l[s:t]=a
```

マージソート

くっつけかた

- l: $\underset{j}{4}, 5, 8, \quad \underset{k}{1}, 2, 6, 7,$
- a:

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $\underset{j}{4}, 5, 8, \quad 1, \underset{k}{2}, 6, 7,$
- a: 1,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $\underset{j}{4}, 5, 8, \qquad 1, 2, \underset{k}{6}, 7,$
- a: 1, 2,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: 4, $\underset{j}{5}$, 8,   1, 2, $\underset{k}{6}$, 7,
- a: 1, 2, 4,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $4, 5, \underset{j}{8}, \qquad 1, 2, \underset{k}{6}, 7,$
- a: 1, 2, 4, 5,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $4, 5, \underset{j}{8}, \quad 1, 2, 6, \underset{k}{7},$
- a: 1, 2, 4, 5, 6,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: 4, 5, 8, 　　1, 2, 6, 7,
  j 　　　　　　　　k
- a: 1, 2, 4, 5, 6, 7,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: 4, 5, 8, $_j$ 1, 2, 6, 7, $_k$
- a: 1, 2, 4, 5, 6, 7, 8

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

クイックソート

- 実用的にはとても高速な方法
  - 適当な pivot を選び、それより大きいものと小さいものに分割
  - それぞれをクイックソート
- 計算量は $O(n \log n)$ (最悪の場合は $O(n^2)$)

```
# あまり早くない実装
def quick_sort(l):
    if len(l)<=1: return
    pivot = random.choice(l)
    le = [i for i in l if i<pivot]
    eq = [i for i in l if i==pivot]
    gr = [i for i in l if i>pivot]
    quick_sort(le)
    quick_sort(gr)
    l[:]=(le+eq+gr)
```

おまけ：線形探索と二分探索

リストに入った指定データがどこにあるか検索する方法

- 線形探索：
  - 先頭から順に調べていく方法
  - 計算量は $O(n)$
- 二分探索：
  - 昇順にソート済みのリストに対し，中央の値を見て，指定データの方が大きければ後半を，小さければ前半を再帰的に調べる．
  - 計算量は $O(\log n)$

```
# 昇順にソート済みのリスト l に対し，要素が x 以下である最大のインデックスを求める
def binary_search(l,x):
    lo, hi = 0, len(l) # [lo,hi) の中に目標はあるとする
    while hi-lo>1: # 答えの候補が 1 つになるまで
        mid = (lo+hi)//2
        if l[mid]<=x: lo=mid
        else: hi=mid
    return lo
```

おまけ2：k 番目に小さい値

ソートされていないデータにおいて k 番目に小さい値を見つける方法

- 昇順にソートして k 番目の値を見る $\Rightarrow O(n \log n)$ 時間
- クイックソートに似た分割統治法 $\Rightarrow O(n)$ 時間

```
def find(l,k):
    n = len(l)
    if n==1: return l[0]
    pivot = random.choice(l)
    le = [i for i in l if i<pivot]
    gr = [i for i in l if i>pivot]
    if len(le)>=k: return find(le,k)
    if n-len(gr)<k: return find(gr,k-(n-len(gr)))
    return pivot
```

計算量は $T(n) = T(n/2) + O(n) \Rightarrow T(n) = O(n)$

アウトライン

itertools

- 今日のプログラミングに有用なモジュール
- 効率的なループ実行のためのイテレータを作る
  - for x in iterator
- 順列，組合せ，直積

```
>>> import itertools
>>> list(itertools.permutations([1,2,3])) # 順列 n!
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
>>> list(itertools.permutations([1,2,3],2)) # 順列 nPk
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> list(itertools.combinations([1,2,3],2)) # 組合せ nCk
[(1, 2), (1, 3), (2, 3)]
>>> list(itertools.product([1,2,3],'ab')) # 直積
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
>>> list(itertools.product([1,2],repeat=2)) # 重複順列
[(1, 1), (1, 2), (2, 1), (2, 2)]
>>> list(itertools.combinations_with_replacement([1,2,3], 2)) # 重複組合せ
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

バックトラック法 (Backtracking)

ブルートフォース（しらみつぶし法，全探索）

- 正しい解を得られるまで可能な組合せを全て試す
- 組合せ爆発を起こすので小さい問題しか解けない

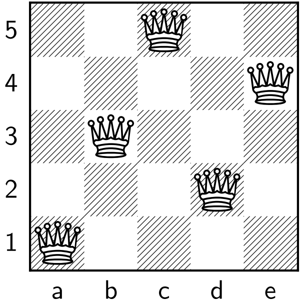
バックトラック

- ブルートフォースを改良した方法
- ダメな候補をある程度ひとまとめに排除する
- 組合せ爆発を抑えられることがある
- 計算量の理論評価をすることは難しい

N クイーン問題

N クイーン問題

$N \times N$ のチェス盤に N 個のクイーンを，互いに縦横斜めの位置にならないように配置せよ．



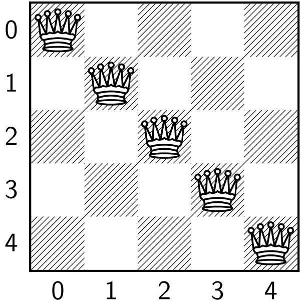
N クイーン問題 — ブルートフォース

方針：適当にクイーンを配置して条件を満たしているかチェックする

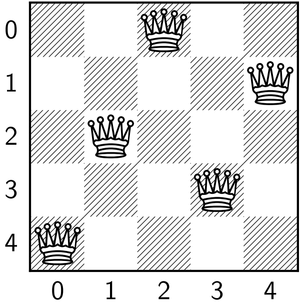
- 条件のチェックは $O(N^2)$ 時間でできる
  (各ペアについてチェックすればよい)
- 各行についてどこにクイーンを置くか決める
  - 適当に配置する $\Rightarrow N^N$ 通り
  - 列が異なるように配置する $\Rightarrow N!$ 通り
    - $10! = 3628800$
    - $11! = 39916800$
    - $12! = 479001600$
  - $N = 10, 11$ くらいまでがんばれそう

N クイーン問題

クイーンの配置を，クイーンの位置の配列として表現



$[0, 1, 2, 3, 4]$



$[2, 4, 1, 3, 0]$

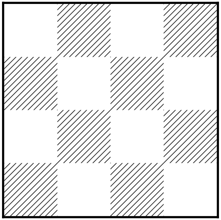
N クイーン問題 — Python によるブルートフォース

```
import itertools

def show(a):
    n = len(a)
    for i in range(n):
        s = ['_']*n
        s[a[i]] = 'Q'
        print(''.join(s))
    print()

def check(a):
    # 各ペアについてチェック
    for (i,j) in itertools.combinations(range(len(a)),2):
            # 同じ列か斜めの関係ならダメ
            if (a[i]==a[j]) or (abs(i-j)==abs(a[i]-a[j])): return False
    return True

def bruteforce(n):
    for a in itertools.permutations(range(n)): # 全列挙
        if check(a): show(a)

bruteforce(8)
```

N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

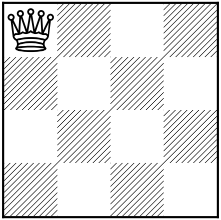
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  ⇒ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

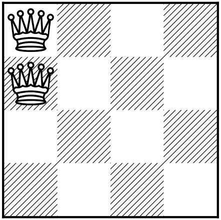
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

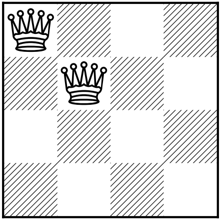
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  ⇒ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

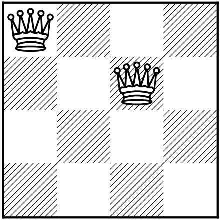
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

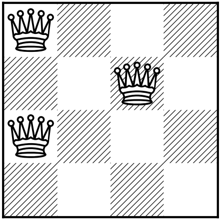
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  ⇒ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

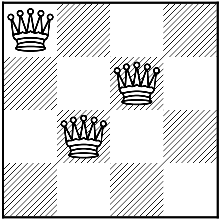
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

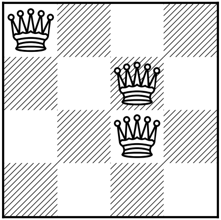
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

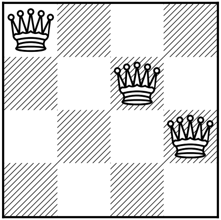
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  ⇒ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

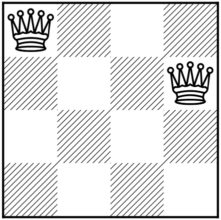
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  ⇒ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

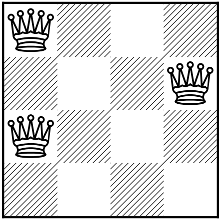
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

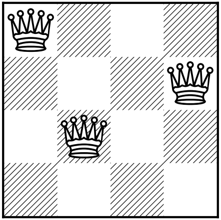
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
$\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

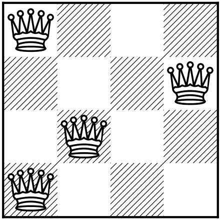
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

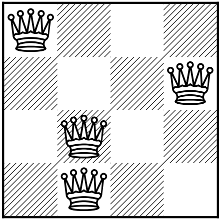
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

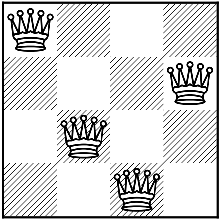
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

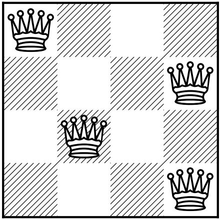
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

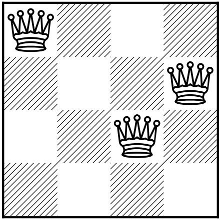
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

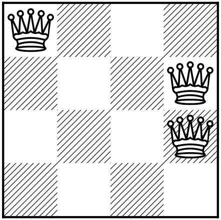
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

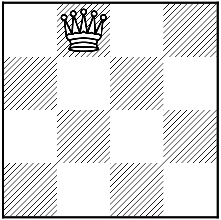
- 途中で条件を満たさなくなったら，残りの行を考える必要はない $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

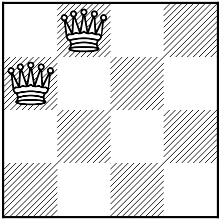
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  ⇒ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

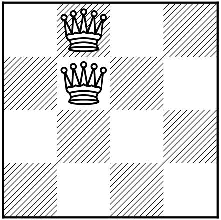
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

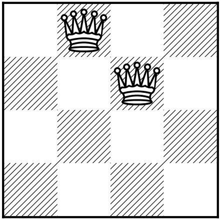
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  ⇒ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



Nクイーン問題 — バックトラック

方針：一行ずつクイーンをおく

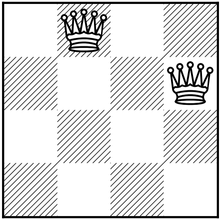
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  ⇒ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

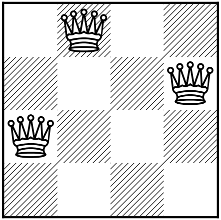
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

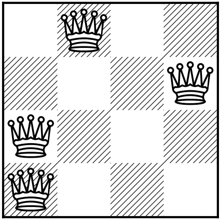
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

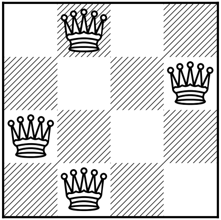
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

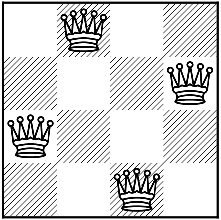
- 途中で条件を満たさなくなったら，残りの行を考える必要はない $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

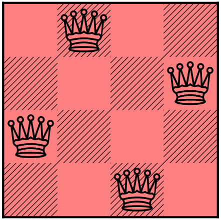
- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  $\Rightarrow$ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — バックトラック

方針：一行ずつクイーンをおく

- 途中で条件を満たさなくなったら，残りの行を考える必要はない
  ⇒ バックトラックする
- 再帰関数を使って実装する (一種の深さ優先探索)
- 計算量の理論評価は難しい... (ブルートフォースよりは効率がよい)



N クイーン問題 — Python によるバックトラック

```
def _backtrack(a,n):
    if len(a)==n: # 完成
        show(a)
        return
    for s in range(n): # 次の行の s 列目においてみる

        # すでに s 列目においてある
        if s in a: continue

        # 斜めの位置においてある
        if any([len(a)-i==abs(a[i]-s) for i in range(len(a))]): continue
        a.append(s)
        _backtrack(a,n)
        a.pop()

def backtrack(n):
    _backtrack([],n)

backtrack(8)
```

数独

- あいているマスに 1 から 9 のいずれかの数字を入れる
- 各行，各列，各ブロックには，同じ数字が複数入ってはいけない

		5	3					
8							2	
	7			1		5		
4					5	3		
	1			7				6
		3	2				8	
	6		5					9
		4					3	
					9	7		

数独

- あいているマスに 1 から 9 のいずれかの数字を入れる
- 各行，各列，各ブロックには，同じ数字が複数入ってはいけない

1	*4*	5	3	*2*	*7*	*6*	*9*	*8*
8	*3*	*9*	*6*	*5*	*4*	*1*	2	*7*
6	7	*2*	*9*	1	*8*	5	*4*	*3*
4	*9*	*6*	*1*	*8*	5	3	*7*	*2*
2	1	*8*	*4*	7	*3*	*9*	*5*	6
7	*5*	3	2	*9*	*6*	*4*	8	*1*
3	6	*7*	5	*4*	*2*	*8*	*1*	9
9	*8*	4	*7*	*6*	*1*	*2*	3	*5*
5	*2*	*1*	*8*	*3*	9	7	*6*	*4*

数独 — Python によるバックトラック

```
def check(table,i,j,k): # table[i][j] に k を入れられるか
    for l in range(9):
        if table[i][l]==k: return False
        if table[l][j]==k: return False
        if table[l//3+(i//3)*3][l%3+(j//3)*3]==k: return False
    return True

def _solve(table,p): # p マス目以降について解く
    if p==81:
        show(table)
        return
    (i,j) = divmod(p,9) # p を行，列に分解
    if table[i][j]: # p マス目に元から数字がある
        _solve(table,p+1)
    else:
        for k in range(1,10): # (i,j) マスに k を入れて続きを考える
            if check(table,i,j,k):
                table[i][j]=k
                _solve(table,p+1)
                table[i][j]=0

def solve(table): _solve(table,0)
```

数独 — 続 Python によるバックトラック

```
def show(table):
    for l in table:
        print(''.join(map(str,l)))
    print()

prob = [[0,0,5,3,0,0,0,0,0],
        [8,0,0,0,0,0,0,2,0],
        [0,7,0,0,1,0,5,0,0],
        [4,0,0,0,0,5,3,0,0],
        [0,1,0,0,7,0,0,0,6],
        [0,0,3,2,0,0,0,8,0],
        [0,6,0,5,0,0,0,0,9],
        [0,0,4,0,0,0,0,3,0],
        [0,0,0,0,0,9,7,0,0]]

show(prob)
solve(prob)
```

アウトライン

演習問題提出方法

解答プログラムをまとめたテキストファイルを作成して，OCW-i で提出

- ファイル名は practice5.txt
- 次回授業の開始時間が締め切り
- ファイルの最初に学籍番号と名前を書く
- どの演習問題のプログラムかわかるように記述
- 出力結果もつける（描画する問題の場合はどのような結果が得られたか一言で説明）
- 途中までしかできなくても，どこまでできてどこができなかったかを書けば部分点を付けます

演習問題 (1/2)

問 1

bubble_sort, merge_sort, quick_sort 以外のソート方法を調べ，どのようなアルゴリズムであるか説明し，実装せよ．
ランダムなリストに対して各ソート関数 (bubble_sort, merge_sort, quick_sort，新しく実装したもの) の実行時間を比較せよ．

問 2

N クイーン問題の解の個数を $N = 8, 9, 10, 11, 12$ について求めよ．

演習問題 (2/2)

問 3

適当に数独の問題をもってきて，プログラムで解け．

問 4 (おまけ)

長さがそれぞれ http://yambi.jp/lecture/advanced_programming2018/prob5-4.txt であるような 1 万本のヒモがある．これらのヒモを切って，同じ長さのヒモを何本かつくることを考える．

1. 長さ 10 万のヒモは最大何本つくれるか．
2. 整数値長さのヒモを 10 万本作るときの最長の長さを求めよ．