Algorithms (2020 Summer)
#5：整列（ソート）

矢谷 浩司

前回の質問

「ローリングハッシュ法でスライド通り実装したつもりなのですがTLEになってしまいます。」

→スライドどおりの基数と除数でテストケースに通るはずです．剰余を取り切れていないところがあると非常に大きな値となることがあり，それが計算速度の足を引っ張ることになります．

前回の質問

「一番初めの照合対象文字列の例では、力任せ法とBM法でやっていることが変わらないと思うのですが、なんでこんなに時間の差が出るのでしょうか」

→BM法でもジャンプができないケースですが，t_iは後戻りしません．力任せ法では，毎回最後の文字まで照合し，失敗したらt_iが後戻りするので，BM法や他のものと比べて，格段に性能が悪くなります．

前回の質問

「ローリングハッシュはある程度の繰り返しを持つ文字列じゃないと使えないということですか？」

→特にそのような制約はありません.

「文字列のハッシュを計算するとき、文字列がアルファベットならば、あらかじめアルファベットに整数インデックスをつけなければいけない、ということですか？」

→はい．何かしらの変換が必要です．課題ではord()を使っていただいたと思いますが，それ以外の変換でも問題はありません.

前回の質問

「リアクションが「手を挙げる」しかないので、リアクションしにくいです。」

→webinarだと手を挙げるしかできないようです．😭勘違いしていて申し訳ありません．私がslackに書き込みをして，そこにリアクションする形で今日はやってみたいと思います．

ソートの典型問題

ランダムな整数が格納されている長さNの配列を昇順・降順に並べ替える.

[3, 5, 2, 1, 6, 4] -> [1, 2, 3, 4, 5, 6]

（以下の説明では上の配列のように昇順にソートすることを前提としますが，降順でも考え方は同じです.）

なぜソート？

人間にとってわかりやすい順序（スプレッドシートの並べ順など）.

探索の時にソートされていることが有利に働く.

実際問題は．．．

ソートに関するライブラリ，関数はどの言語でも大概充実しているので，それを使うのがよい．

非常に効率的に動くように実装されているので，自前の関数でやるメリットはあまりない．．．

使えるメモリ量も十分なことが多いので，ソートのやり方による領域計算量の影響が小さくなった．

とはいえ，その中身を理解することはとても重要．

内部 vs. 外部

ソート実行時に一時的に必要な記憶領域が元々のデータ量以上，つまり$O(n)$以上の場合，外部ソートと呼ぶ．
（ディスク等の外部の記憶領域が必要になるようなイメージ）

一方，$O(1)$や$O(\log n)$の一時的な記憶領域が必要なものは内部ソートと呼ぶ．（なんとなくメモリ上で対応できる，というイメージ）

安定 vs. 安定でない

同一の要素が複数ある場合，最初の並び順がソート完了後も保持されている場合，「安定」という．

安定でないソートアルゴリズムでもソートはちゃんとされるが，同一要素間での並びは変わってしまう場合あり．

要素を飛び越えて入れ替えるようなアルゴリズムの場合，安定でなくなる．

安定 vs. 安定でない

例）[3, 1, 4, 2, 5, 3]

安定なソート：必ず[1, 2, 3, 3, 4, 5]
安定でないソート：[1, 2, 3, 3, 4, 5]になることがある

要素の出てくる順番が重要になるかどうか.

超非効率ソート: ボゴソート

bogus + sort = bogosort

完全運任せソート

与えられた列をランダムにシャッフルし，たまたま完全にソートされている状態になるまで繰り返す.

超非効率ソート: ボゴソート

列をシャッフルした後，たまたまソートされた状態になっているという場合に遭遇するためには平均$n!/2$回の試行が必要.

さらに，ソートされているかどうかをチェックするために毎回平均$n/2$の比較が必要.

よって，$O(n!\,n)$という素晴らしいほどに非効率なアルゴリズム．（かつ不安定）

挿入ソート（Insertion sort）

与えられた配列の内，頭からi番目まではソートされているとする．

そのときに，i+1番目の要素を入れる場所を順番にチェックして探す．

これを頭から最後尾の要素まで繰り返す．

挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]

挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] そのまま

挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] そのまま
#2：[3, 5, 2, 1, 6, 4] そのまま

挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] そのまま
#2：[3, 5, 2, 1, 6, 4] そのまま
#3：[3, 5, 2, 1, 6, 4] -> [2, 3, 5, 1, 6, 4]

挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] そのまま
#2：[3, 5, 2, 1, 6, 4] そのまま
#3：[3, 5, 2, 1, 6, 4] -> [2, 3, 5, 1, 6, 4]
#4：[2, 3, 5, 1, 6, 4] -> [1, 2, 3, 5, 6, 4]

挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] そのまま
#2：[3, 5, 2, 1, 6, 4] そのまま
#3：[3, 5, 2, 1, 6, 4] -> [2, 3, 5, 1, 6, 4]
#4：[2, 3, 5, 1, 6, 4] -> [1, 2, 3, 5, 6, 4]
#5：[1, 2, 3, 5, 6, 4] そのまま

挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] そのまま
#2：[3, 5, 2, 1, 6, 4] そのまま
#3：[3, 5, 2, 1, 6, 4] -> [2, 3, 5, 1, 6, 4]
#4：[2, 3, 5, 1, 6, 4] -> [1, 2, 3, 5, 6, 4]
#5：[1, 2, 3, 5, 6, 4] そのまま
#6：[1, 2, 3, 5, 6, 4] -> [1, 2, 3, 4, 5, 6]

挿入ソートの計算量

1回の平均的な比較・移動回数は$i/2$. それをn回繰り返すので, $\sum_i^n i/2 \to O(n^2)$.

最悪のケースはどんな場合？

挿入する場所を探す際に，二分探索を使うこともできる.

挿入ソート

わかりやすい！実装も単純.

追加の記憶領域をほとんど必要としない.

事前にソートされている配列に追加するときには有利.

安定的アルゴリズムとして実装できる.

バブルソート

並べたい順に1つずつ「浮き上がらせていく」ソート.

水の中の気泡が浮き上がってくるようなイメージ.

より小さい値をどんどん配列の前に送っていく.

バブルソート

（以下，配列の要素を順番に後ろから見ていく場合．）

今見ている要素（n番目）が1つ前の要素（n-1番目）より小さい場合，場所を入れ替える．

同じことをn-1番目とn-2番目の要素について行い，以降繰り返す．

最後まで行うと，1番目の要素は一番小さい値になる．

バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]

バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]

バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]

#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]

#2：[3, 5, 2, 1, 4, 6] そのまま

バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]
#2：[3, 5, 2, 1, 4, 6] そのまま
#3：[3, 5, 2, 1, 4, 6] -> [3, 5, 1, 2, 4, 6]

バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]
#2：[3, 5, 2, 1, 4, 6] そのまま
#3：[3, 5, 2, 1, 4, 6] -> [3, 5, 1, 2, 4, 6]
#4：[3, 5, 1, 2, 4, 6] -> [3, 1, 5, 2, 4, 6]

バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]
#2：[3, 5, 2, 1, 4, 6] そのまま
#3：[3, 5, 2, 1, 4, 6] -> [3, 5, 1, 2, 4, 6]
#4：[3, 5, 1, 2, 4, 6] -> [3, 1, 5, 2, 4, 6]
#5：[3, 1, 5, 2, 4, 6] -> [1, 3, 5, 2, 4, 6]

これで，1の場所は確定．次は[3, 5, 2, 4, 6]を処理．以下，同じように続ける．

バブルソートの計算量

こちらも先ほどと似たような感じで，$O(n^2)$．よって速くはないが，コードがとてもシンプル．

ただし，比較の回数が必ず$n(n-1)/2$回になるので，挿入ソートよりも少し遅くなる（交換の回数は挿入ソートと同じ）．

こちらも安定的．

シェイカーソート（Cocktail shaker sort）

双方向からやるバブルソート.

1つの方向からバブルソートし，最後まで行ったら今度は逆方向に進める.

最後にスワップを行った場所から逆方向のバブルソートを開始する.

シェイカーソート例

初期状態：[3, 5, 2, 1, 6, 4]

後ろから前にバブルソート.
[3, 5, 2, 1, 6, 4] -> … -> [1, 3, 5, 2, 4, 6]

一番最後のスワップはindex：1の場所（3）.

シェイカーソート例

次は，前から後ろにバブルソート．
[1, 3, 5, 2, 4, 6] -> [1, 3, 2, 5, 4, 6] -> [1, 3, 2, 4, 5, 6]

一番最後のスワップはindex：3の場所（4）．

また逆方向からバブルソート．以降これを繰り返す．

シェイカーソート実装例

```
def shakersort(seq):

    # ソート済みの左端，右端を保持する変数
    right = len(seq) - 2
    left = 0

    # 最後にスワップが起きた場所を格納する変数
    swapped = 0
```

シェイカーソート実装例

```
def shakersort(seq):
    ...
    while left < right:
        for i in range(left, right): # 先頭からチェック
            if seq[i+1] < seq[i]:
                seq[i], seq[i+1] = seq[i+1], seq[i]
                swapped = i
        # 最後のスワップの場所でrightを更新
        right = swapped
```

シェイカーソート実装例

```
def shakersort(seq):
    ...
    while left < right:
        ...
        [次は後ろからチェック]
        [最後のスワップの場所でleftを更新]


        # このwhileループ1回で左右からのチェック
        # を済ませることになる.
```

バブルソートと何が違う？

もし，1方向動くときに最後に連続してk回スワップがなかった場合，そのk個分の要素はすでにソートされていることになる.

よって，その分は次回以降考えなくて良いことになる.
（最後にスワップした場所を覚えておく理由）

この分だけバブルソートよりもちょっと効率が良い.

シェイカーソートの計算量

最悪計算量はこちらも$O(n^2)$.

ただし，バブルソートよりはちょっと速いことが期待できる.

ノームソート（Gnome sort）

妖精（ノーム）が植木鉢を入れ替えてソートするところからこの名前がついている.

バブルソートと似ているが，スワップしたら前に進むか，後ろに進むかが変わる.

スワップしなかった：前に1つ進む

スワップした：後に1つ戻る

ノームソート例

初期状態：[3, 5, 2, 1, 6, 4]

[3, 5, 2, 1, 6, 4] -> スワップなしなので，前に1つ進む.

ノームソート例

初期状態：[3, 5, 2, 1, 6, 4]

[3, 5, 2, 1, 6, 4] -> スワップなしなので，前に1つ進む．
[3, 5, 2, 1, 6, 4] -> スワップありなので，後ろ1つ戻る．

ノームソート例

初期状態：[3, 5, 2, 1, 6, 4]

[3, 5, 2, 1, 6, 4] -> スワップなしなので，前に1つ進む．
[3, 5, 2, 1, 6, 4] -> スワップありなので，後ろ1つ戻る．
[3, 2, 5, 1, 6, 4] -> スワップあり．
（この場合は先頭なので，前に1つ進む）．

ノームソート例

初期状態：[3, 5, 2, 1, 6, 4]

[3, 5, 2, 1, 6, 4] -> スワップなしなので，前に1つ進む．
[3, 5, 2, 1, 6, 4] -> スワップありなので，後ろ1つ戻る．
[3, 2, 5, 1, 6, 4] -> スワップあり．
[3, 2, 5, 1, 6, 4] -> スワップなしなので，前に1つ進む．

ノームソート例

初期状態：[3, 5, 2, 1, 6, 4]

[3, 5, 2, 1, 6, 4] -> スワップありなので，前に1つ進む．
[3, 5, 2, 1, 6, 4] -> スワップありなので，後ろ1つ戻る．
[3, 2, 5, 1, 6, 4] -> スワップあり．
[3, 2, 5, 1, 6, 4] -> スワップなしなので，前に1つ進む．
[3, 2, 5, 1, 6, 4] -> スワップありなので，後ろ1つ戻る．

以降，繰り返す．

ノームソート実装例

```
def gnomesort(seq):
    i = 0
    while i < len(seq)-1:
        if i == -1: i = 1
        elif seq[i+1] >= seq[i]: i += 1
        else:
            seq[i], seq[i+1] = seq[i+1], seq[i]
            i -= 1
    return seq
```

ノームソートの計算量

最悪計算量はこちらも$O(n^2)$.

実装ではループを1重にでき，単純に実装できる.

ただし，，，

今までに紹介したものは多少の差はあれど，$O(n^2)$.

まだまだ遅い．．．

なんとかしようぜ.

与えられた配列をそのまま扱うのでは無理がある.

じゃあどうする？

- 処理で工夫する
- データ構造で工夫する

分割統治法（divide and conquer）

1つをまとめて処理するのは大変．．．

そこで領域をすぐに処理できるレベルまで小分けにして，
そこで処理する．

それを順にくっつけて戻していけば，最終的に達成したい
ゴールにたどり着く．

クイックソートとマージソートが代表例．

クイックソート

その名の通り，速い！（ごく例外的なケースを除く）

Tony Hoareさんによって考案.

Hoare, C. A. (1962). Quicksort. *The Computer Journal*, *5*(1), 10-16.

クイックソートの考え方

配列の中から1つ値を選ぶ（枢軸とよぶ）.

枢軸の選び方にはいろいろあるが，とりあえず今与えられている配列の真ん中に位置する要素とする.

クイックソートの考え方

枢軸より小さいものと大きいものにソートされていなくても良いので，振り分ける．

振り分けた後，2つのグループに分割し，各グループに対して同じ処理を行う．最終的に要素1つのグループになる．

それらを全部結合すれば終わり！

クイックソート例

[7, 8, 4, 5, 6, 2, 3, 1]　　# 5を枢軸にする

クイックソート例

端からスタート.

[7, 8, 4, 5, 6, 2, 3, 1]

クイックソート例

7は枢軸5よりも大きく，1は枢軸5よりも小さい．

[7, 8, 4, 5, 6, 2, 3, 1]

よって，この2つを入れ替える．

[1, 8, 4, 5, 6, 2, 3, 7]

クイックソート例

[1, 8, 4, 5, 6, 2, 3, 7]

左右のカーソルを1つ進める.

[1, 8, 4, 5, 6, 2, 3, 7]

クイックソート例

8は枢軸5よりも大きく，3は枢軸5よりも小さい．

[1, 8, 4, 5, 6, 2, 3, 7]

よって，この2つを入れ替える．

[1, 3, 4, 5, 6, 2, 8, 7]

クイックソート例

[1, 3, 4, 5, 6, 2, 8, 7]

左右のカーソルを1つ進める.

[1, 3, 4, 5, 6, 2, 8, 7]

クイックソート例

2は枢軸5より小さいのでスワップの候補になる．しかし，
4も枢軸5より小さいので，これはスワップしたくない．

[1, 3, 4, 5, 6, 2, 8, 7]

クイックソート例

そこで左側のみカーソルを1つ進める.

[1, 3, 4, 5, 6, 2, 8, 7]

5は枢軸と同じ（これは枢軸より大きいとみなす）なので,
スワップを行う.

[1, 3, 4, 2, 6, 5, 8, 7]

クイックソート例

左右のカーソルを1つ進めると，2つのカーソルが交差することになる．これで，この段階は終了．

[1, 3, 4, 2, 6 , 5, 8, 7]

この時点では左側には枢軸より小さいもの，右側には枢軸より大きいものが集まっている（ただし，この時点ではソートされていない）．

クイックソート例

[1, 3, 4, 2]と[6, 5, 8, 7]に分割！

次のステップでは，この分割したものに対してクイックソートを個別に行う（ということは実装的にはどうすればよい？）

クイックソートの実装例

```
def qsort(seq (配列), left (左端index), right (右端index)):
    [再帰の終了条件は？]

    pivot = [与えられた範囲の真ん中に位置する要素]
    # カーソルを用意
    cur_l = left
    cur_r = right
```

クイックソートの実装例

```
def qsort(seq, left, right):
    ...
    while True:
        [左カーソル（cur_l）をpivotより大きい要素まで進める]
        [右カーソル（cur_r）をpivotより小さい要素まで進める]
```

クイックソートの実装例

```
def qsort(seq, left, right):
        …
        while True:
                …
                # カーソル交差でループ終了
                if (cur_r <= cur_l): break
```

クイックソートの実装例

```
def qsort(seq, left, right):
        ...
        while True:
                ...
                # cur_l < cur_rだが，左側にpivotより大きい要素，
                # 右側にpivotより小さい要素がそれぞれ見つかった．
                [cur_rの要素とcur_lの要素を入れ替える]
```

クイックソートの実装例

```
def qsort(seq (配列), left (左端index), right (右端index)):
    …

    while True:
        …

    # このwhileループを抜けるということは，左右の
    # 要素の入れ替えが完了したこと.
```

クイックソートの実装例

```
def qsort(seq (配列), left (左端index), right (右端index)):
    …
    while True:
        …
    # 分割したグループそれぞれに対して，qsortを
    実行したい！ -> 再帰
    [左側だけでqsort]
    [右側だけでqsort]
    # どんな引数を与えれば良い？
```

クイックソートの実行例

seq = [3, 8, 14, 12, 90, 1, 4, 29, 43 , 2, 10, 6, 37, 78, 50, 18]
quicksort(seq, 0, len(seq)-1)
-----実行結果-----
3, 8, 14, 12, 18, 1, 4, 6, 10, 2, 43, 29, 37, 78, 50, 90

クイックソートの実行例

seq = [3, 8, 14, 12, 90, 1, 4, 29, 43 , 2, 10, 6, 37, 78, 50, 18]

quicksort(seq, 0, len(seq)-1)

-----実行結果-----

3, 8, 14, 12, 18, 1, 4, 6, 10, 2, 43, 29, 37, 78, 50, 90

3, 8, 14, 12, 2, 1, 4, 6, 10, 18, 43, 29, 37, 78, 50, 90

クイックソートの実行例

seq = [3, 8, 14, 12, 90, 1, 4, 29, 43 , 2, 10, 6, 37, 78, 50, 18]

quicksort(seq, 0, len(seq)-1)

-----実行結果-----

3, 8, 14, 12, 18, 1, 4, 6, 10, 2, 43, 29, 37, 78, 50, 90

3, 8, 14, 12, 2, 1, 4, 6, 10, 18, 43, 29, 37, 78, 50, 90

1, 2, 14, 12, 8, 3, 4, 6, 10, 18, 43, 29, 37, 78, 50, 90

クイックソートの実行例

seq = [3, 8, 14, 12, 90, 1, 4, 29, 43 , 2, 10, 6, 37, 78, 50, 18]

quicksort(seq, 0, len(seq)-1)

-----実行結果-----

3, 8, 14, 12, 18, 1, 4, 6, 10, 2, 43, 29, 37, 78, 50, 90

3, 8, 14, 12, 2, 1, 4, 6, 10, 18, 43, 29, 37, 78, 50, 90

1, 2, 14, 12, 8, 3, 4, 6, 10, 18, 43, 29, 37, 78, 50, 90

1, 2, 3, 12, 8, 14, 4, 6, 10, 18, 43, 29, 37, 78, 50, 90

…

1, 2, 3, 4, 6, 8, 10, 12, 14, 18, 29, 37, 43, 50, 78, 90

クイックソートの計算量

一般的には，$O(n \log n)$．よってかなり効率的！

枢軸の選び方によっては$O(n^2)$になる（グルーピングがどちらか一方に完全に偏る場合）が，一般的にはそう頻発しない．

安定的ではない（要素を飛び越えてスワップするため）．

クイックソートの計算量

$O(n \log n)$の前提は，与えられる配列要素の並び方は$1/n!$で等確率である．

実際のデータにはある種の偏りがあることも．

枢軸をランダムに選ぶことで，意地悪いケースでもうまく対応できる．（乱択化）

マージソート

分割統治法による代表的なもう1つのアルゴリズム.

与えられた配列を2分割していき，要素1個の配列まで小さくする.

分割したもの同士をソートをしながら結合して，元の配列の大きさに戻す.

フォン・ノイマンによる考案（1945年）とされている.

マージソートの実装例

```
def mergesort(seq):
    if len(seq) <= 1: return seq

    # 2つに分割するだけ
    left = mergesort(seq[:len(seq)//2])
    right = mergesort(seq[len(seq)//2:])

    # これらの値が返ってきたときには，left，right
    # 各々でソートができている状態になっている．
```

マージソートの実装例

```
def mergesort(seq):
        …
        merged = []
        cur_l = cur_r = 0
        # マージ作業．小さい方から順にmergedに入れる．
        while cur_l < len(left) and cur_r < len(right):
                if left[cur_l] <= right[cur_r]:   # 安定性を確保
                        merged.append(left[cur_l])
                        cur_l += 1
                else:
                        merged.append(right[cur_r])
                        cur_r += 1
```

マージソートの実装例

```
def mergesort(seq):
    ...
    # もし余った要素があればくっつける.
    if cur_l < len(left):
        merged.extend(left[cur_l:])
    elif cur_r < len(right):
        merged.extend(right[cur_r:])

    return merged
```

マージソートの計算量

[長さnの配列のソート]
= 2 * [長さn/2の配列のソート] + [n個の要素のマージ]

大雑把に議論すると，長さが1の配列から元の長さにたどり着くまでには$\log n$段階存在する．各段階におけるマージ操作は全部合わせて$O(n)$.

よって，$O(n \log n)$.

マージソートの計算量

最悪の場合での計算量も$O(n \log n)$なので，クイックソートよりも運の悪さに依存しない.

マージ操作の部分があるため，クイックソートよりは一般的には少し遅い.

メモリを食いやすいので，大きい配列のときは注意.

クイックソートとマージソート

クイックソート

分割するときに（ざっくり）ソートする

結合するときは何も考えない

マージソート

分割するときは何も考えない

結合するときにソートする

なんとかしようぜ. （再掲）

与えられた配列をそのまま扱うのでは無理がある.

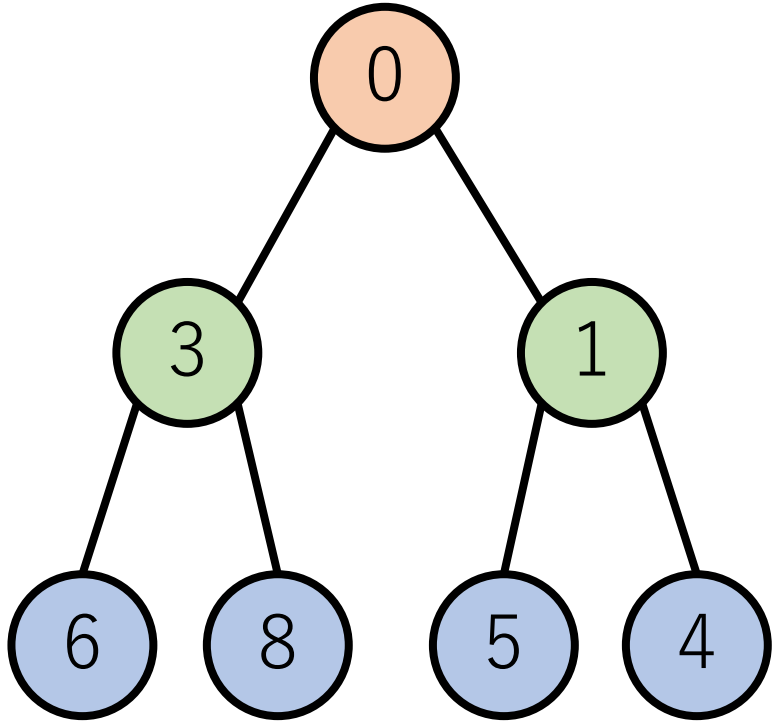
じゃあどうする？

処理で工夫する

データ構造で工夫する

ヒープソート

前に習ったヒープを使おう．ヒープは
根が最大もしくは最小になっている構造.



与えられた配列を全部ヒープに押し
込めた後，1つずつ取り出せば自動的に
ソートされている！

（ヒープに関しては2回目の講義参照）．

ヒープソートの実装例

```
import heapq

def heapsort(seq):
    heap = []
    while seq:          # ヒープを作る
        heapq.heappush(heap, seq.pop())
    while heap:         # ヒープから取り出す
        seq.append(heapq.heappop(heap))
```

ヒープソートの計算量

ヒープを作る：

要素追加ごとに $O(\log n)$の処理が必要.
それが n回起きる.

ヒープから取り出す：

要素削除ごとに$O(\log n)$の処理が必要.
それが n回起きる.

よって, $O(n \log n)$.

ヒープソートの特徴

メリット：

ヒープを使うので，データの出現パターンにあまり影響されない．最悪の場合でも$O(n \log n)$.

デメリット：

ヒープ処理の分があるため，クイックソートよりは一般的には遅い.

記憶領域$O(n)$が必要（ヒープ分）.

それでもまだ$O(n \log n)$. . .

もっと早くできないだろうか？

今までの一般的なソートに対応するアルゴリズム.
　　大小の比較ができれば使える.

条件を限定してより速くできない？

バケットソート（bucket sort）

バケツソート，ビンソートなどとも．

整列したいデータの取りうる値がk種類である前提．

あらかじめk種類の「バケツ」を用意しておく．
与えられた配列をバケツに振り分ける．
振り分け後，整列したい順序でバケツから順番に取り出す．

バケットソート

例）[3, 2, 1, 1, 3]

1，2，3に対応するバケツを用意.

1	2	3

バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく.

1	2	3
		3

バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく.

1	2	3
	2	3

バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく.

1	2	3
1	2	3

バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく．

1	2	3
1, 1	2	3

バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく.

1	2	3
1, 1	2	3, 3

バケットソート

例）[3, 2, 1, 1, 3]

1	2	3
1, 1	2	3, 3

バケツから所望の順序で取り出す.

[1, 1, 2, 3, 3]

バケットソート

バケツは可変長リスト（線形リストなど）で実装.

もし不安定でもよければ，各バケツに対応する値の出現回数のみを記録しておき，その情報を基に出力すべき配列を作り出す.

特にこの実装のものを計数ソート（counting sort）とも呼ぶ.

計数ソート実装例

```
# 0からmax_valueまでの整数値のみと想定
def countsort(seq, max_value):
        count = [0]*(max_value+1)       # バケツ
        sorted = []         # ソート済み配列

        # 出現回数をカウント
        for i in range(len(seq)):
                count[seq[i]] += 1
```

計数ソート実装例

```
def countsort(seq, max_value):
        …
        # 出現回数からソート済み配列を生成
        for i in range(len(count)):
                for j in range(count[i]):
                        sorted.append(i)

        return sorted
```

バケットソート

先の計数ソートの例では，出てくる要素とバケツに付随する値（キー）が一致しているが，そうでなくてもよい．

例えば，a -> 1，b -> 2などでもよい．

バケットソートの計算量

配列の長さがn，出てくる要素の種類（バケツの数）をkとすると，

バケツの準備：一般的には，$O(k)$.
バケツに入れる：$O(n)$.
バケツから取り出す：$O(n)$.

よって，$O(n+k)$.　$k \leq n$なので，　$O(n)$.

バケットソートの計算量

メリット

速い！$O(n)$. 😊

デメリット

強い制約が存在（整列したいデータの取りうる種類が予めわかっている）.

取りうる種類が多い場合，空間計算量的には損することも.

実行時間比較例（ランダムな整数の配列）

[msec]	n=50	n=500	n=5,000
挿入ソート	0.14	9.60	960
バブルソート	0.28	25.2	2,040
シェイカーソート	0.26	20.5	1,722
ノームソート	0.40	40.3	3,670
クイックソート	0.14	0.85	10.7
マージソート	0.23	1.92	21.9
計数ソート	0.08	0.31	2.50

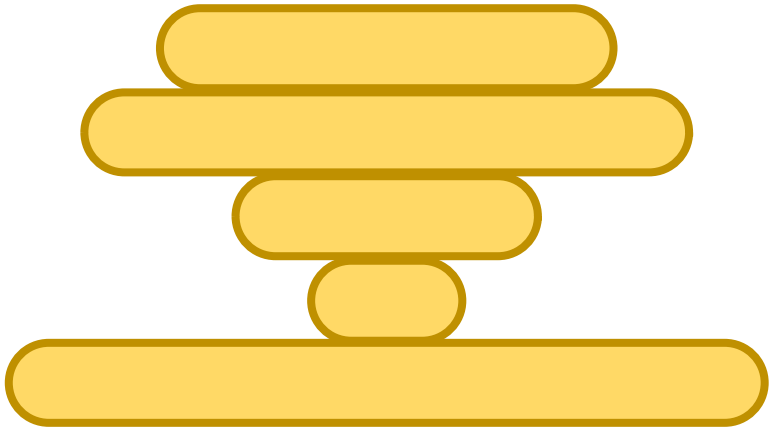
パンケーキソート（余談）

n段のパンケーキを下から大きい順に並べたい.

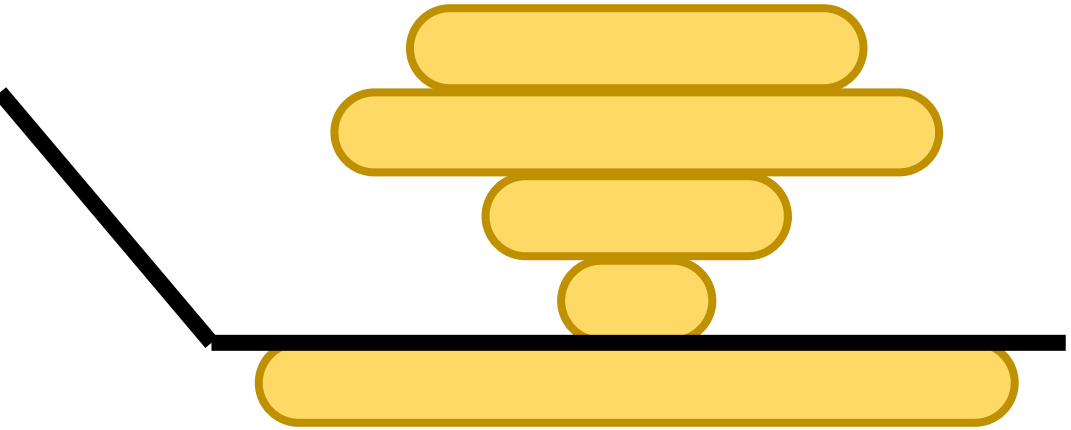
ただし，行える操作は上からi段目のパンケーキに
フライ返しを入れて，一気にひっくり返すことだけ.

配列の「prefixの回転」のみ行える.

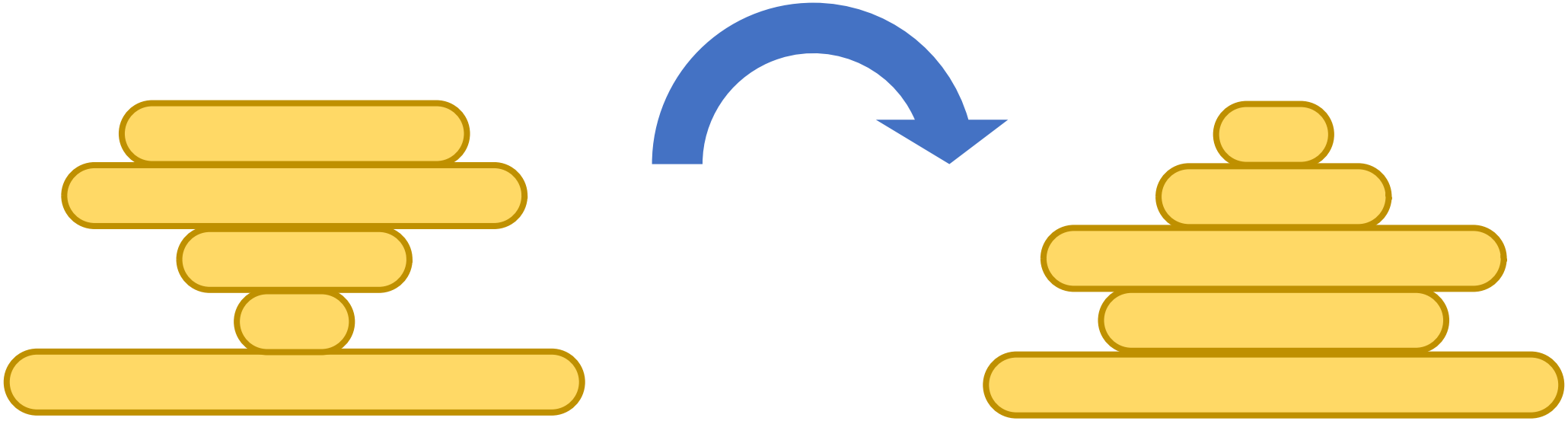
パンケーキソート（余談）



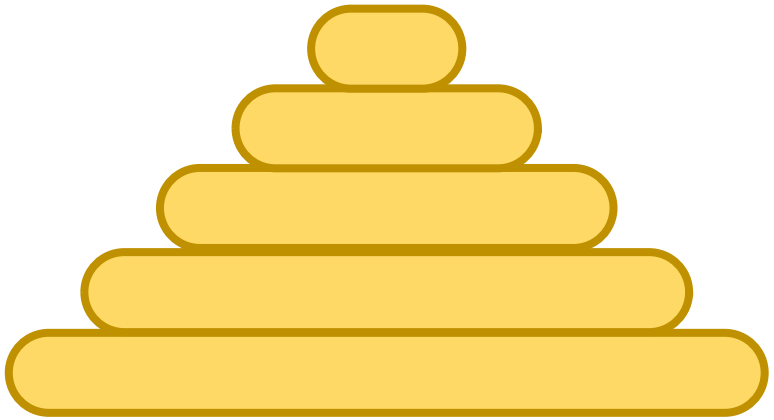
パンケーキソート（余談）



パンケーキソート（余談）

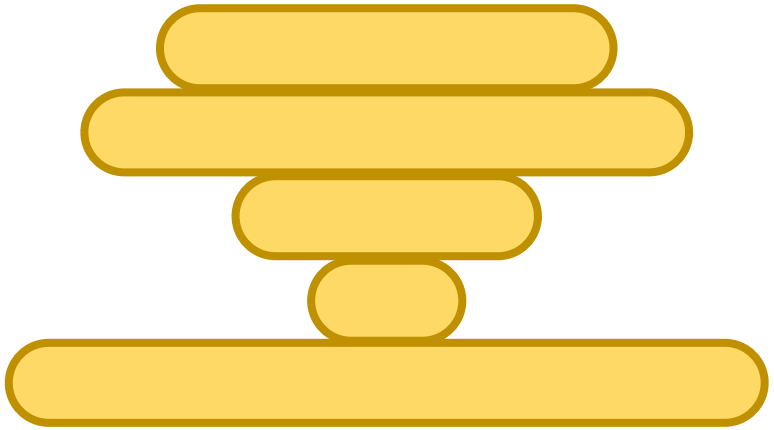


パンケーキソート（余談）



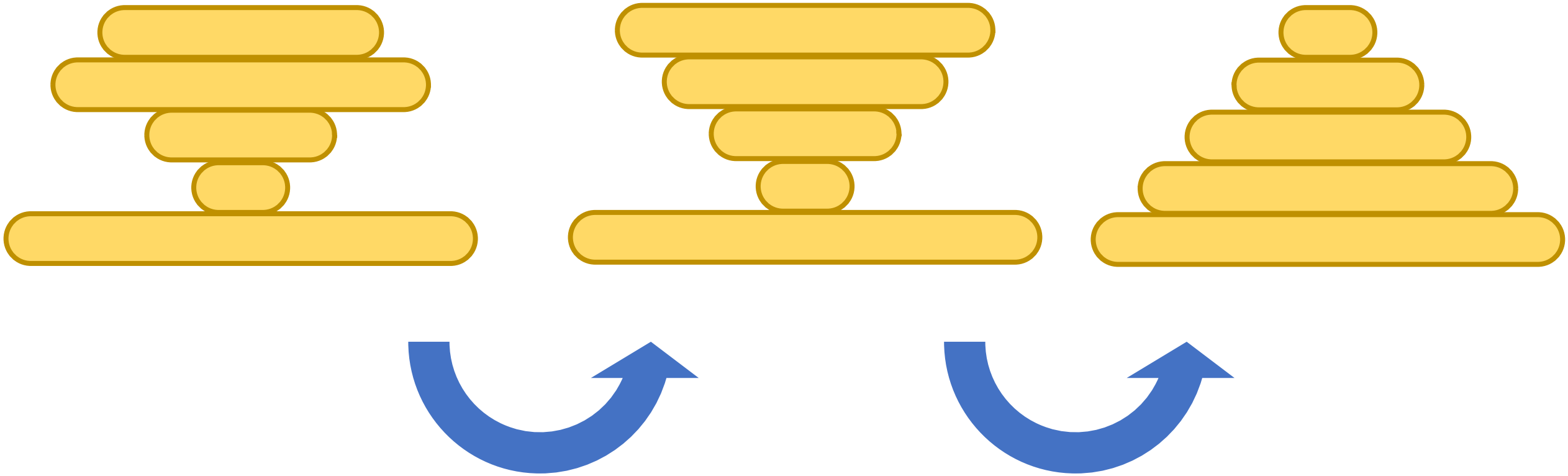
最終的にはこういう状態にしたい.

パンケーキソート（余談）

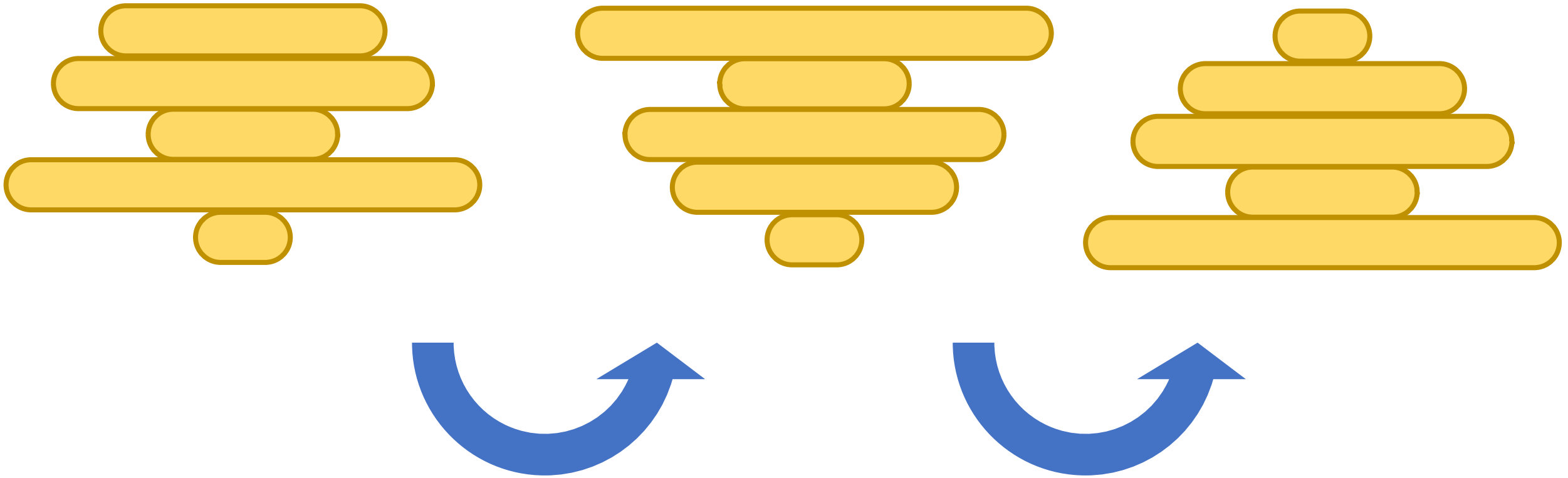


この初期状態の場合には最小何回でソートできるでしょうか？

パンケーキソート（余談）



確実にやるなら



ソートされていないパンケーキの中で一番大きいものの直下から一度ひっくり返し，更にひっくり返す．

ただし，

この方法では各パンケーキに対して2回の操作を行う（一旦一番上まで持ってきて，もう一度ひっくり返す）．

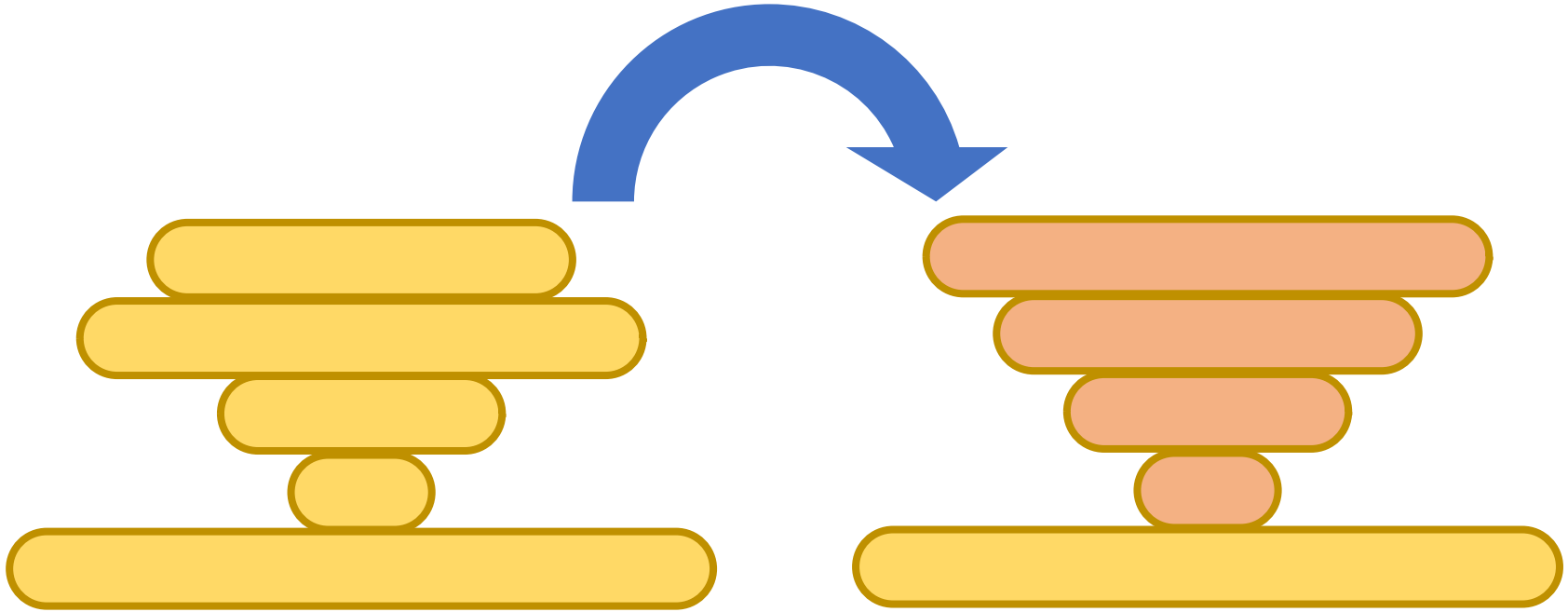
よって，単純に考えると，操作回数の上界は$2n$.

計算量ではなくパンケーキを返す回数であることに注意．「パンケーキを返す」は「配列の要素を入れ替える」ことに対応するので，1回返すと$O(n)$．よって，全体では$O(n^2)$になります．

実際には

もう少し効率的に出来る（先程2回でできたのが例）．

ひっくり返したときにソートされた塊ができる場合には
それを優先する．それ以外では，上の基本のやり方で
ひっくり返す．



実際には

これにより操作回数の上界が$(5n+5)/3$になることが，
知られている．

ということを見つけて，論文に書いた人が皆さんも御存知の方です．どなたでしょう？😊

Discrete Mathematics 27 (1979) 47–57.

BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES
Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation σ of the integers from 1 to n, let $f(\sigma)$ be the smallest number of prefix reversals that will transform σ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all σ in (the symmetric group) S_n. We show that $f(n) \leqslant (5n+5)/3$, and that $f(n) \geqslant 17n/16$ for n a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2-1 \leqslant g(n) \leqslant 2n+3$.

https://people.eecs.berkeley.edu/~christos/papers/Bounds%20For%20Sorting%20By%20Prefix%20Reversal.pdf

まとめ

$O(n^2)$のアルゴリズム

挿入ソート，バブルソート，
シェーカーソート，ノームソート，（パンケーキソート）

$O(n \log n)$のアルゴリズム

クイックソート，マージソート
ヒープソート

$O(n)$のアルゴリズム

バケットソート（使える条件に注意！）

コードチャレンジ：基本課題#5-a [1.5点]

スライドを参考にしながら，バブルソートを実装してください．

コードチャレンジ：基本課題#5-b [1.5点]

スライドを参考にしながら，クイックソートを実装してください．

コードチャレンジ：Extra課題#5 [3点]

うまく並び替えて，総乗の値を最小にする問題.