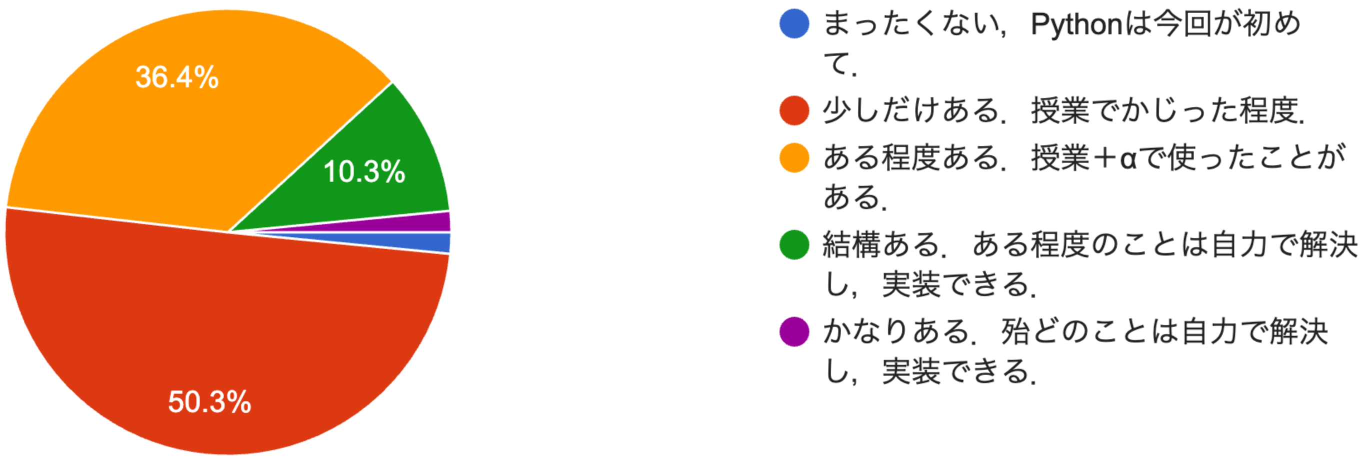
Algorithms (2024 Summer)
#2：累積和，整数関連

矢谷 浩司
東京大学工学部電子情報工学科

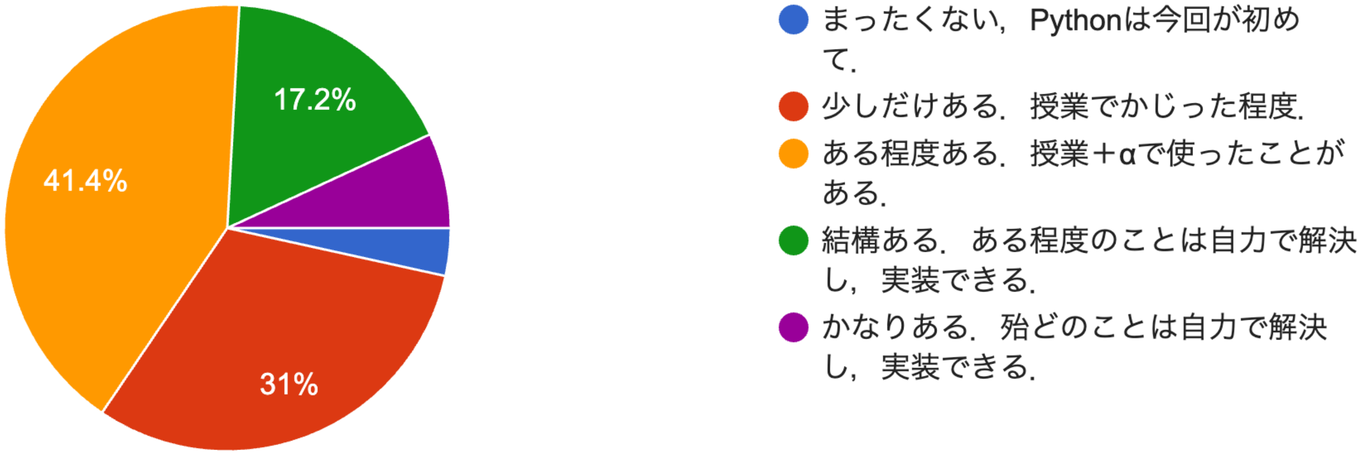
Pythonでのプログラミングにどのくらい自信がありますか？

195 responses



本学学生

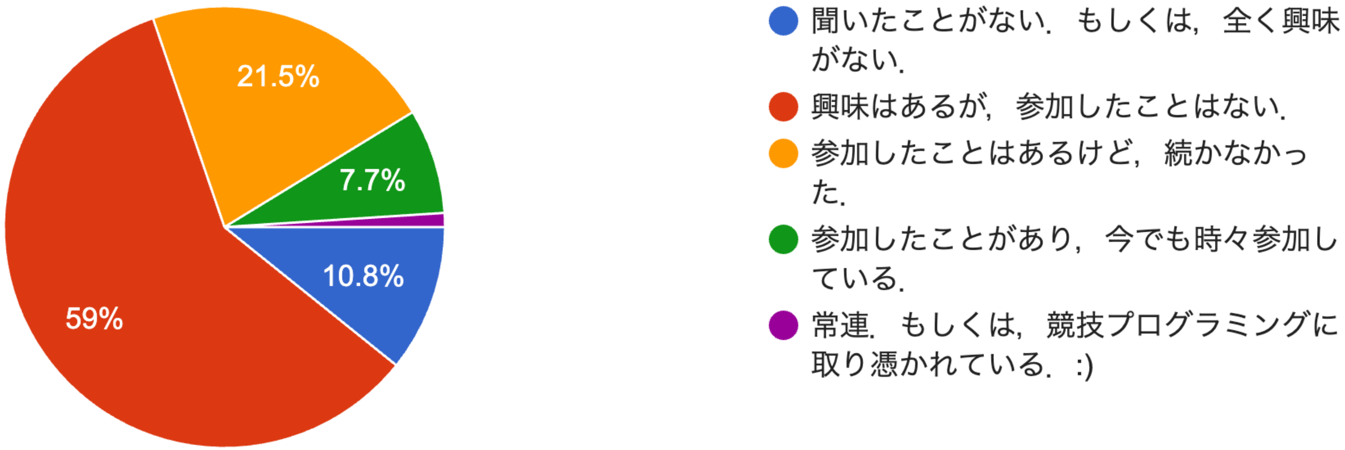
29 responses



メタバース工学部生

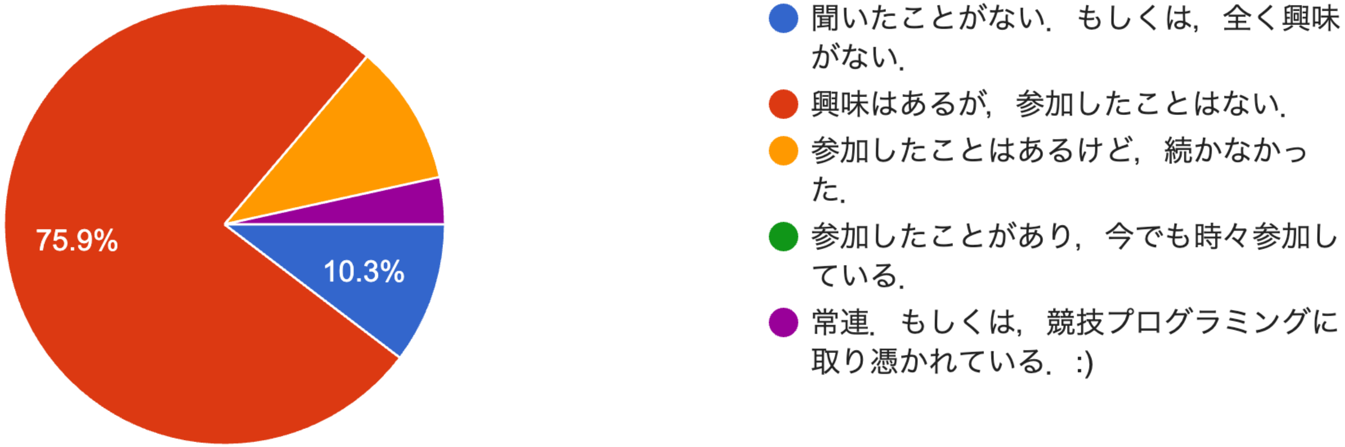
競技プログラミング等に参加したことはありますか？

195 responses



本学学生

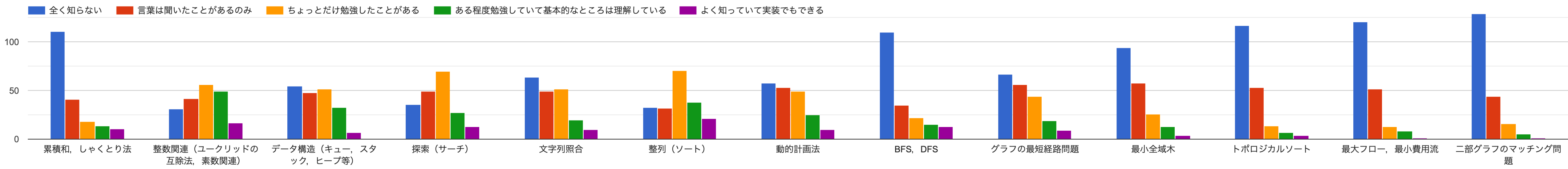
29 responses



メタバース工学部生

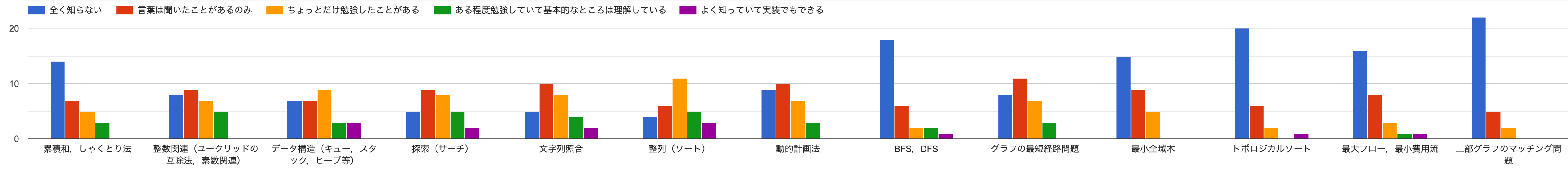
本学学生

アルゴリズムに関していままでにどの程度勉強したことがありますか？



メタバース工学部生

アルゴリズムに関していままでにどの程度勉強したことがありますか？



今日から本格的に進みます！

わからないことや疑問に思うことがあれば，各回のチャネル（今日はclass-02）に投稿してください．随時私かTAさんが拾ってお答えします．☺️

また，私からも授業中に問いかけを行いますので，適宜リアクションをください．よろしくお願いいたします．

しゃくとり法と累積和

しゃくとり法と累積和

しゃくとり法

現在計算している部分和の区間において，条件に応じて右端を伸ばしたり，左端を縮めたりしながら，与えられた配列を左から順にたどりながら計算

累積和

先頭の要素からj番目の要素までの部分和を計算した配列を事前に作り，それを利用してi番目からj番目までの部分和を定数回で計算

しゃくとり法と累積和

しゃくとり法

現在計算している部分和の区間において，条件に応じて右端を伸ばしたり，左端を縮めたりしながら，与えられた配列を左から順にたどりながら計算

累積和

先頭の要素からj番目の要素までの部分和を計算した配列を事前に作り，それを利用してi番目からj番目までの部分和を定数回で計算

前回の基本課題: しゃくとり法の典型問題

ランダムな整数が格納されている長さNの配列の中で,
m個の隣接する要素の和が最大となる部分を1つ求めよ.

（累積和でも解けます.）

ナイーブな解き方

```
[sum_max: 今までの部分和の最大値を入れる]
[max_index: sum_maxになる最初のインデックス]

[i: 0からN-mまでループ]:
    tmp = 0
    [j: 0からm-1までループ]:
        tmp += sequence[i+j]
    if tmp > sum_max:
        [sum_maxとmax_indexを更新]
```

ナイーブな解き方

```
[i: 0からN-mまでループ]:                # N-m+1回のループ
    tmp = 0
    [j: 0からm-1までループ]:            # m回のループ
        tmp += sequence[i+j]
    if tmp > sum_max :
        [sum_maxとmax_indexを更新]
```

$N > m \gg 1$なら，$O(Nm)$. （もしくは，$O(m(N-m))$）
N, mがそれなりに大きいと結構大変.

改良案

毎回 sequence[i] から sequence[i+m-1] まで足し合わせているのが無駄.

その次sequence[i+1] から sequence[i+m]まで計算することになるが，変更があるのは最初と最後だけ.

よってその差分だけ計算するようにすれば無駄を大きく削減できる！

改良版：しゃくとり法

```
tmp = [0からm-1までのsequenceの部分和]    # 最初の部分和
m = tmp
m_index = 0

[i: 0からN-m-1までループ]:    # N-m回のループ
    tmp = tmp - sequence[i] + sequence[m+i]
    if tmp > m:
        [mとm_indexを更新]
```

改良版：しゃくとり法

新しい部分和を計算するところが$O(m)$から$O(1)$に.

よって，全体の計算量も$O(N)$！

パフォーマンスの比較

N=10,000,　m=100
（表の単位はmsec）

ナイーブ方式：$O(Nm)$
改良版：$O(N)$

ナイーブ方式	改良版
206	3.3
163	4.4
191	4.3
158	3.3
167	3.4
170	3.4
158	3.5
170	3.4
163	3.5
170	3.7

パフォーマンスの比較

N=100,000,　m=100
Nを10倍

ナイーブ方式：$O(Nm)$
改良版：$O(N)$

どちらもほぼ10倍になる.

ナイーブ方式	改良版
1,638	37
1,616	37
1,614	37
1,624	38
1,642	38
1,624	36
1,617	38
1,662	36
1,662	42
1,643	36

パフォーマンスの比較

N=100,000,　m=1,000
さらにmを10倍.

ナイーブ方式： $O(Nm)$
改良版： $O(N)$

ナイーブ方式には影響するが,
改良版には影響しない.

ナイーブ方式	改良版
17,676	37
19,122	51
20,810	61
19,720	41
20,130	40
20,147	41
18,543	46
20,981	38
20,167	40
19,770	38

しゃくとり法でもう1問

「ランダムな非負整数が格納されている長さNの配列の中で，部分和がmになる連続した要素のうち，その長さが最小になるものを求めよ．」

先ほどの問題と違い，<u>長さが可変</u>.

今回のしゃくとり法の考え方

部分和を計算する左端，右端のindexを保持する変数を定義．それぞれ0からスタート．

右端のindexを1つずつ増やしながら部分和を計算．

決められた値（例の場合はm）を部分和が超えたら，右端のindexを増やすのをやめる．

今回のしゃくとり法の考え方

次に，左端のindexを進めながら部分和を更新（つまり最初の要素から順に部分和から引いていく）．

決められた値（例の場合はm）を部分和を下回ったら超えたら，左端のindexを増やすのをやめる．

また右端のindexを右に動かしていき，以降同様に繰り返す．ぴったりmになった時には現在の部分和を構成する連続するようその長さを比較し，最短なら記録しておく．

ピンとこない時は

とか言われても．．．😅

アルゴリズムの説明を聞かされてよくわからない時は,
とりあえず小さな例（1桁くらいの入力サイズ）で
手で動かしてみる.

手で紙に書いてみるのが，役に立ちます！

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3, 14より下 -> 右端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3,　14より下 -> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は7,　14より下-> 右端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3，14より下 -> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は7，14より下-> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は16，14より上-> 左端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3, 14より下 -> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は7, 14より下-> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は16, 14より上-> 左端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は13, 14より下-> 右端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す．

[3, 4, 9, 5, 1, 4, 6]: 部分和は3，14より下 -> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は7，14より下-> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は16，14より上-> 左端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は13，14より下-> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は18，14より上-> 左端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3, 14より下 -> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は7, 14より下-> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は16, 14より上-> 左端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は13, 14より下-> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は18, 14より上-> 左端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は14, ぴったり！

しゃくとり法の操作

今考えている区間において，

　所望の値に足りない→右端を伸ばす

　所望の値を超えている→左端を縮める

このルールが揺るがないので，しゃくとり虫が前に進むように，伸び縮みしながらも必ず前に進む．

　右端，左端は動くタイミングは違うが，最終的にはどちらの端も左から右まで後戻りせずに動く．

しゃくとり法の前提

右端を伸ばす時，左端を縮める時，広義の単調増加・減少性が存在することが前提条件となる．

「広義」というのは0増える（減る）のを許すこと．

つまり，今考えている値は右端を伸ばしたら0以上増える，左端を縮めたら0以上減る，が必ず実現されている．

もし，右端を伸ばして値が減るケースがあると，どちらの端を操作するべきか判断できなくなってしまう．

しゃくとり法の計算量

区間の選び方の総数は，$n(n+1)/2$.

左端と右端の間に含まれるのが1つの要素だけのときも考える必要があるので，$n+1$個選べる場所がある.

よって，総当りでやると計算量は$O(n^2)$.

しゃくとり法では，左端も右端もn回しか動かず，かつ，部分和の更新は定数回.

よって，計算量は$O(n)$. 右端が後戻りしない分効率化できている.

しゃくとり法と累積和

しゃくとり法

現在計算している部分和の区間において，条件に応じて右端を伸ばしたり，左端を縮めたりしながら，与えられた配列を左から順にたどりながら計算

累積和

先頭の要素からj番目の要素までの部分和を計算した配列を事前に作り，それを利用してi番目からj番目までの部分和を定数回で計算

累積和の考え方

あらかじめ先頭からj番目までの和の計算結果を保持する配列を作っておく.

例）[1, 2, 3, 4, 5]

s[0] = 1　　　# 0番目のみ
s[1] = 3　　　# 0番目から1番目の和
s[2] = 6　　　# 0番目から2番目の和
s[3] = 10　　 # 0番目から3番目の和
s[4] = 15　　 # 0番目から4番目の和

累積和を使った区間和の計算

i番目（i>0）からj番目までの和が欲しい時

　　→s[i-1]とs[j]の値を使って差を計算

例）[1, 2, 3, 4, 5]で1番目から3番目の和が欲しい.

　　s[3] - s[0] = 9

配列の要素の呼び出しと引き算だけで実現でき，
定数回の処理で済む！

累積和の典型問題：区間和

あるデータのうちAからBまでで該当するデータの数や和を問い合わせるクエリが大量に発生する，みたいなシナリオ．

例）「A月からB月までの総売上を問い合わせるクエリが大量に発生する．」

毎回，指定された区間の和を計算していると大変．

クエリの数Q，区間の長さ平均mで，計算量$O(Qm)$．

累積和の典型問題：区間和

累積和であらかじめ最初の月からi月までの和を計算.

→配列の長さNならば，$O(N)$.

ある月からある月までの和を求める.

→どんな区間を指定されても引き算だけで実行できるので，$O(1)$.

この場合，クエリの数Qとすると，全体としては計算量$O(N+Q)$.

前回の基本課題: しゃくとり法の典型問題

ランダムな整数が格納されている長さNの配列の中で,
m個の隣接する要素の和が最大となる部分を1つ求めよ.

（累積和でも解けます．）

累積和での別解法

長さNの配列に対して，0番目からi番目までの和を保持する配列s[i]を作る.

とりあえず，0番目からm-1番目までの部分和を最大として記録しておく.

　　→これはs[m-1].

0からN-mのjに対して，s[m+j]-s[j]を順にチェックし，より大きな部分和が出ればそれを記録する.

累積和での別解法の計算量

長さNの配列に対して，0番目からi番目までの和を保持する配列s[i]を作る．

$\rightarrow O(N)$

0からN-mのjに対して，s[m+j]-s[j]を順にチェックし，より大きな部分和が出ればそれを記録する．

$\rightarrow O(N)$

よって，全体としても$O(N)$.

そのほかの問題でも

「ランダムな非負整数が格納されている長さNの配列の中で，部分和がmになる連続した要素のうち，その長さが最小になるものを求めよ．」の問題も累積和で解くことができます．

累積和の配列sに対して，しゃくとり法をするようなイメージ．

計算量は$O(N)$だが，累積和を計算した上で，しゃくとり法することになるので，定数倍効率は悪い．

まとめ

しゃくとり法と累積和は似ているところがありますので，一緒に理解できるといいと思います． 😄

Web上では混同して使われることも．

これらの考え方はこの後の講義でもちらほら出てきますので，是非覚えておいてください．

累積和は2次元に拡張したものもありますので，興味のある方は勉強してみてください．

大きさ[N, M]の表において[i, j]から[k, l]までの全てのセルの和を求める，みたいな問題．

整数関連

お次のお題は

整数関連の処理についてのお話.

あまり授業では取り上げられないトピックですが,
皆さんに馴染みがある整数であることと，紹介する
アルゴリズムはどれもシンプルなので，ご紹介します．

競技プログラミングで役に立つ基礎知識でもあります． 😊

最大公約数

「2つの整数a，ｂの最大公約数を求めよ．」

例）

14と30 -> 2

786,240と76,608 -> 4032

単純なやり方

2から順に調べる？

最悪のケースでは$O(\min(a, b))$かかる.

ユークリッドの互除法

「$a, b\ (a \geq b)$の最大公約数は$a\%b$（剰余）とbの最大公約数に等しい．」

明記されている最古のアルゴリズムだそうです.
（紀元前300年くらい）

a，bの剰余を交互に繰り返して行き，どちらかが0になった時点で終了.

ユークリッドの互除法

238と8736の最大公約数を求める．

ユークリッドの互除法

238と8736の最大公約数を求める.

8736 % 238 = 168

238 % 168 = 70

168 % 70 = 28

70 % 28 = 14

28 % 14 = 0

ユークリッドの互除法

238と8736の最大公約数を求める.

8736 % 238 = 168

238 % 168 = 70

168 % 70 = 28

70 % 28 = 14 <- これが最大公約数となる.

28 % 14 = 0

ユークリッドの互除法の実装例

```
def gcd(a, b):
    if b==0:
        return a
    else:
        # 値の小さい方が常に2番目に来るようにする
        return gcd(b, a%b)
```

ユークリッドの互除法の計算量

1回のgcdでa, bのどちらかは半分以下になる.

よって大まかには$O(\log(\max(a, b)))$かかる.

厳密にはラメの定理により，小さい方の整数の桁数の5倍が上限であることが知られている.

ちなみに最悪のケースはどんなもの？

最小公倍数

最大公約数がわかれば，　$a * b/gcd(a, b)$で求める．

（言語によっては，数字が大きい場合は計算順序に注意する必要あり．）

例）

14と30 -> 210

786,240と76,608 -> 14,938,560

拡張ユークリッドの互除法

一次不定方程式の整数解の1つを求める.

例）$14x + 6y = 4$ -> $x = 2, y = -4$

拡張ユークリッドの互除法

一次不定方程式$\mathrm{a}x + by = c$が整数解を持つ必要十分条件はcが$\mathrm{gcd(a, b)}$で割り切れることである.

（証明はここでは省略. . . ）

つまり, $c = d * \mathrm{gcd(a, b)}$となるので, $\mathrm{a}x + by = \mathrm{gcd(a, b)}$を計算できれば, 元の式に対する答えもわかる.

拡張ユークリッドの互除法

$14x + 6y = 4$の例で考えてみると，$14x + 6y = 2$を解けば良い．

つぎに14と6の最大公約数をユークリッドの互除法を使って求めると，

14, 6 -> 2, 6 -> 2, 0

になる．

拡張ユークリッドの互除法

$14x + 6y = 2$を分解すると，$(12 + 2)x + 6y = 2$であるから，$2x + 6(y + 2x) = 2$となる．

つまり，　$2x + 6y' = 2$を解けば，$y = y' - 2x$から元の解が求まる．

さらに$2(x + 3y') + 0y' = 2 \ \rightarrow 2x' = 2$を解けば，元の解が求まる．

この場合，　$x' = 1, y' = 0, x = 1, y = -2$と順々に求まる．

拡張ユークリッドの互除法

ユークリッドの互除法を利用することで，x, yの係数をどんどん小さくすることが出来，最終的には明示的に求まる形になる．

再帰を使って実装すれば良い．

拡張ユークリッドの互除法の実装例

```
# gcd(a, b), x, yが返る.
def ext_gcd(a, b):
    if b == 0:
        return a, 1, 0
    else:
        d, x, y = ext_gcd(b, a%b)
        return d, y, x - (a//b)*y
```

拡張ユークリッドの互除法の計算量

こちらも大まかには$O(\log(\max(a,b)))$.

素数判定

「与えられた整数nが素数であるかを判定せよ．」

例）

13 -> Yes

25 -> No

1,000,000,007（10^9 + 7）-> Yes

素数判定

ナイーブな方法

2からn/2まで順番に割っていき，割り切ることができればNo．そうでなければYes.

この場合，計算量は$O(n)$.

素数判定

もし，　dがnの約数だとすると，n/dもnの約数.
（例：n=30, d=3なら，n/d=10もnの約数）

つまり，　$(d, n/d)$が必ずペアになっている.

$\min(d, n/d)$の最大値は$\sqrt{n}$となるので，2から$\sqrt{n}$まで調べれば良いことになる.

こうすると，計算量が$O(\sqrt{n})$まで削減できる.

素数判定アルゴリズムの実装例

```
def prime(n):
    if n <= 1: False
    i = 2
    while i*i <= n:  # √nまで繰り返す
        if n%i == 0: return False
        i += 1
    return True
```

素数数え上げ

「1以上n以下の素数の数を求めよ．」
「1以上n以下の素数を全て求めよ．」

例）
13 -> 6（2, 3, 5, 7, 11, 13）
25 -> 9（2, 3, 5, 7, 11, 13, 17, 19, 23）
1,000,000 -> 78498

素数判定アルゴリズムを単純に使うと

1からnまで素数判定を繰り返す．よって，計算量は
$O(\sum_{i=1}^{n}\sqrt{i})$.

$$\int_1^n \sqrt{x}\,dx = \int_1^n x^{\frac{1}{2}}\,dx = \frac{2}{3}\left(n^{\frac{3}{2}} - 1\right)$$

であることを考えれば，全体の計算量は$O(n\sqrt{n})$.
（単純に，$O(\sqrt{n})$がn回あると考えても良い．）

nが大きいと，ちょっと遅い．．．

改良案

エラトステネスの篩というアルゴリズムを使い，素数のリストを作ってから，数を数える．

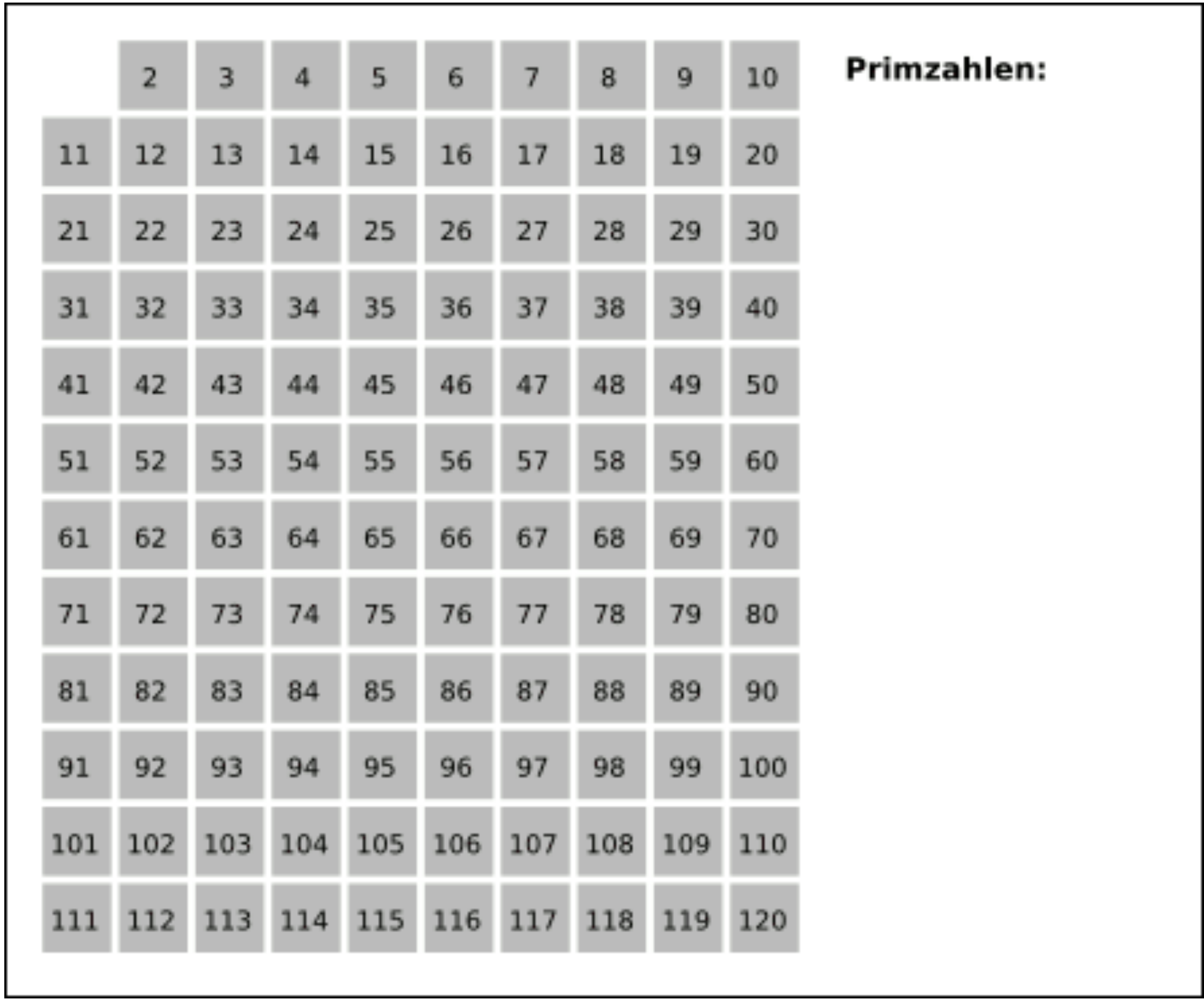
エラトステネスの篩もユークリッドの互除法くらい古いアルゴリズムとされている．

（「エラトス・テネス」と発音するのがギリシャ語に近いらしいです．）

エラトステネスの篩

#1 2からスタート.
#2 2の倍数を全部削除.
#3 次の数字（3）に移る.
#4 3の倍数を全部削除.
#5 次の数字（5）に移る.
#6 5の倍数を全部削除.

以降，$\sqrt{n}$まで繰り返す.



https://ja.wikipedia.org/wiki/%E3%82%A8%E3%83%A9%E3%83%88%E3%82%B9%E3%83%86%E3%83%8D%E3%82%B9%E3%81%AE%E7%AF%A9

エラトステネスの篩の実装例

```
def prime_all(n):
    is_prime = [True]*(n+1)   # 素数かどうかのリスト
    i = 2         # 0, 1は無視し，2からスタート
```

エラトステネスの篩の実装例

```
def prime_all(n):
        is_prime = [True]*(n+1)
        i = 2
        while i*i <= n:  # √nまで繰り返す

                [素数の倍数に関して，is_primeを全て
                Falseにする]
```

エラトステネスの篩の実装例

```
def prime_all(n):
    is_prime = [True]*(n+1); i = 2
    while i*i <= n:
        if is_prime[i]:    # iが素数ならば
            [iの倍数でn以下の値は全て素数でないと記録]
        i += 1      # 次の整数へ
```

エラトステネスの篩の実装例

```
def prime_all(n):
    is_prime = [True]*(n+1); i = 2
    while i*i <= n:
        if is_prime[i]:
            [iの倍数でn以下の値は全て素数でないと記録]
        i += 1
    # 2以上でTrueのものの個数を返す
    return len([i for i in range(2, n+1) if is_prime[i]])
```

エラトステネスの篩の実行例

prime_all(25)

9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
実行前	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

エラトステネスの篩の実行例

prime_all(25)

9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
実行前	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
i = 2	T	T	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T

エラトステネスの篩の実行例

prime_all(25)

9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
実行前	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
i = 2	T	T	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
i = 3	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T

エラトステネスの篩の実行例

prime_all(25)

9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
実行前	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
i = 2	T	T	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
i = 3	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
i = 4	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T

エラトステネスの篩の実行例

prime_all(25)

9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
実行前	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
i = 2	T	T	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
i = 3	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
i = 4	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
i = 5	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F

エラトステネスの篩の実行例

prime_all(25)

9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
実行前	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
i = 2	T	T	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
i = 3	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
i = 4	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
i = 5	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F
終了時	T	T	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F

エラトステネスの篩の計算量

```
def prime_all(n):
    is_prime = [True]*(n+1); i = 2
    while i*i <= n:
```

<- $\sqrt{n}$まで繰り返すから，$O(\sqrt{n})$？

```
        if is_prime[i]:
            [iの倍数でn以下の値は全て素数でないと記録]
        i += 1
    return len([i for i in range(2, n+1) if is_prime[i]])
```

エラトステネスの篩の計算量

```
def prime_all(n):
    is_prime = [True]*(n+1); i = 2
    while i*i <= n:
        if is_prime[i]:
            [iの倍数でn以下の値は全て素数で
            ないと記録]    # ここの更新が効いてくる
        i += 1
    return len([i for i in range(2, n+1) if is_prime[i]])
```

エラトステネスの篩の計算量

2で篩に落とされるのはn/2個.
3で篩に落とされるのはn/3個.
5で篩に落とされるのはn/5個.
…

よって,

$$n\left(\frac{1}{2}+\frac{1}{3}+\frac{1}{5}+\frac{1}{7}+\cdots+\frac{1}{\sqrt{n}}\right)$$

個振り落とす（Falseにする）ことになる.

エラトステネスの篩の計算量

$$\frac{1}{2}+\frac{1}{3}+\frac{1}{5}+\frac{1}{7}+\cdots+\frac{1}{\sqrt{n}} \approx \log\log\sqrt{n}$$

ということが知られている（nが十分に大きければ）．
よって，振り落とす数の総和は$n\log\log\sqrt{n}$くらいになるので，計算量は$O(n\log\log n)$.

実質的には$O(n)$となる.

素因数分解

与えられた数nを，2から$\sqrt{n}$まで順番に割っていき，
1になれば終了.

割り切れたらその数を記録.

割り切れなくなるまでその数で割り続ける.

そうでなければ次の数に行く.

$\sqrt{n}$までいっても与えられた数が1になっていなければ，
その数も入れる.　（nが素数の場合）

このやり方の場合，　$O(\sqrt{n})$.

SPFを利用する素因数分解

Smallest Prime Factor（SPF）が事前に分かっていると，
高速化できる．

SPF：1以外の最小の素因数

例）12 -> 2, 15 -> 3

SPFを利用する素因数分解

エラトステネスの篩のコードを少し変更し，事前にnまでのSPFを求めるようにする.

例）spf_table = spf(13)

[0, 1, 2, 3, 2, 5, 2, 7, 2, 3, 2, 11, 2, 13]

spf_table[n]がnのSPFを保持している（spf[0]とspf[1]は無視する）.

SPFを利用する素因数分解

```
def spf(n):
    spf = [_ for _ in range(n+1)]; i = 2
    while i*i <= n:
        if spf[i] == i:
            j = 2*i
            while j <=n:
                if spf[j] == j: spf[j] = i
                j += i
        i += 1
return spf
```

SPFを利用する素因数分解

あとはこのspf_tableを逐次参照し，割っていけば良い.

```
def PrimeFactorization(n):
        spf_table = spf(n)
        while n > 1:
                print(spf_table[n])
                n  //= spf_table[n]
```

SPFを利用する素因数分解

SPFの事前計算は，エラトステネスの篩と同じく，
$O(n \log\log n)$.

素因数分解の処理は，毎回2以上の値で割ることができるので，$O(\log n)$.

もし素因数分解を求めるクエリがQ回くるような場合，

ナイーブな方法だと，$O(Q\sqrt{n})$.

SPFを使う方法だと，$O(n \log\log n + Q \log n)$.

幂乗

「x^nを求めよ．」

まともに計算すると$O(n)$.
（オーバーフローが起きないと仮定）

繰り返し自乗法

$$x^n = (x^2)^{\frac{n}{2}}$$

と変形すれば，計算回数を半分にできる！

これを再帰で繰り返せば，$O(\log n)$で計算可能.

繰り返し自乗法

x**nでよくね？😅

繰り返し自乗法

x**nでよくね？😅

pythonだとそれでもたまたま扱えるだけで，言語によってはオーバーフローや型の変換等を考える必要が出てくる．

pythonでも値が大きくなりすぎると計算が大変になる．

剰余を求める

よって，32bit（10進数で10桁）で収まるようにするために，大きな素数の剰余を使い，それを計算に使う．

競技プログラミングなどでは「1000000007（10の9乗+7）で割った余りを求めよ．」という指示があることが多い．

この手の問題では計算過程でも値が大きくなることがあり，計算過程中でもmod (10**9 +7)を計算する必要が出てくる．

剰余を取りながら冪乗を計算する際には，先程の繰り返し自乗法が役に立つ．

剰余で答えを出す場合

加算：

加算したあとでmod hを計算.

減算：

減算したあとでmod hを計算. （ただし，言語によっては計算結果が負のときにはhを足す必要あり. ）

乗算：

乗算したあとでmod hを計算.

剰余で答えを出す場合

(-20) % 7を実行すると，

Python：1

C++：-6

C++の場合，さらに7足すと，1となり一致する．

繰り返し自乗法の擬似コード

```
def power(x, n):
        M = 10**9 + 7
        if n==0: return 1        # 0乗は1.

        [(x^2)^(n/2)を再帰で計算]        # 剰余はどこで取る？
        [nが奇数の時の追加処理]        # 何をすれば良い？

        [計算結果を返す]
```

$(x^2)^{\frac{n}{2}}$

繰り返し自乗法（剰余で答えを出す場合）

```
def power(x, n):
    M = 10**9 + 7
    if n==0: return 1        # 0乗は1.

    tmp = power(x*x % M, n//2)    # 再帰で計算
    if n%2: tmp = tmp * x % M     # nが奇数の場合の処理

    return tmp
```

繰り返し自乗法の実行結果例

power(7, 3)（$\mathbf{7^3}$）-> 343

power(7, 30) （$\mathbf{7^{30}}$）-> 157,445,110

もし剰余を取らないと，$\mathbf{7^{30}}$は
22,539,340,290,692,258,087,863,249
とかいうものすごい値に．．．

剰余で答えを出す場合（再掲）

加算：

加算したあとでmod hを計算.

減算：

減算したあとでmod hを計算.　（ただし，言語によっては計算結果が負のときにはhを足す必要あり．）

乗算：

乗算したあとでmod hを計算.

剰余で答えを出す場合（再掲）

加算：

加算したあとでmod hを計算.

減算：

減算したあとでmod hを計算. （ただし，言語によっては計算結果が負のときにはhを足す必要あり. ）

乗算：

乗算したあとでmod hを計算.

除算：

？

加算，減算，乗算はかんたん．

これら3つの場合は，剰余を余計にとっても問題ない．

例）9*2 = 18 でmod 7をとる．

mod 7を最後だけ取る．

9*2 = 18 -> 18 mod 7 = 4

毎回mod 7を取る．

9 mod 7 = 2, 2 mod 7 = 2 -> 2*2 = 2 -> 4 mod 7 = 4

除算はちょっと厄介．．．

除算はそうはいかない．．．

例）$8/2 = 4$ でmod 7をとる．

mod 7を最後だけ取る．

$8/2 = 4 \quad \to \quad 4 \bmod 7 = 4$

毎回mod 7を取る．

$8 \bmod 7 = 1,\ 2 \bmod 7 = 2 \to \ 1/2 = 0.5$

$\to 0.5 \bmod 7 = 0.5$??

除算はちょっと厄介．．．

計算の最後だけで剰余を取るのであれば大丈夫.

だけど，そこに至るまでにすでに剰余を取ってしまっている場合には計算が狂ってしまう．．．

フェルマーの小定理

aが任意の自然数，mが素数でa,mが互いに素である時，
以下のことが成立する．

$$a^{m-1} \equiv 1 \bmod m$$

逆元

mが素数で，aがmでは割り切れない整数であるとき，
以下の式を満たすxが$[1, \mathrm{m})$ の範囲で一意に存在する．
このようなxを「$\mathrm{mod}\, m$におけるaの逆元（逆数のより一般的なもの）」と呼ぶ．

$$ax \equiv 1 \bmod m$$

つまり，

通常の世界：aに掛けると1になる数 -> $1/a$
$\mathrm{mod}\, m$ の世界：aに掛けると1になる数 -> x

この2つを使うと，

$a^{m-1} \equiv 1 \bmod m$は， $a \times a^{m-2} \equiv 1 \bmod m$と見ることが出来る．つまり， aの逆元は$\bmod m$の世界ではa^{m-2}になる．

$b \div a$は$b \times (1 \div a)$と変形できることから，$\bmod m$の世界では，aの逆元がわかれば割り算を計算できる！

つまり， **aで割ることはa^{m-2}をかけることに$\bmod m$の世界では等しい**，ということになる．

先ほどの例を計算し直してみよう.

「8/2 = 4 でmod 7をとる」は, 「8 * 2^(7-2) でmod 7をとる」に読み替えて計算できるはず.

mod 7を最後だけ取る.

8/2 = 4　->　4 mod 7 = 4

毎回mod 7を取る.

8 mod 7 = 1, 2^(7-2) mod 7 = 4 ->　1 * 4 = 4

-> 4 mod 7 = 4

フェルマーの小定理

aが任意の自然数，mが素数でa, mが互いに素である時,
以下のことが成立する.

$$a^{m-1} \equiv 1 \bmod m$$

mが素数でない場合，逆元が存在しないこともあるので注意.
例えば$a = 2, m = 6$だと，$2^{(6-1)} \bmod 6 = 2$となり,
フェルマーの小定理が成立していないことがわかる.

組み合わせの計算

「${}_nC_k$の998244353で割った余りをもとめよ．」

$$ {}_nC_k = \frac{n!}{(n-k)!\,k!} $$

だが，$n!$をまともに計算してしまうと大変な数字になるので，剰余で計算していく必要がある．では分母をどう処理するか？

組み合わせの計算

$$\frac{n!}{(n-k)!\,k!} \mod M = n!\,((n-k)!)^{M-2}(k!)^{M-2} \mod M$$

と変形すれば計算できる．

よって，剰余を取りながら上の計算を行えば良い．

まとめ

しゃくとり法，累積和

みなさんもぜひスライドを見ながら追実装し，
性能比較してみてください．

しゃくとり法・累積和的な考え方は今後ちらほら
出てきます．

整数関連

ユークリッドの互除法，素数判定，エラトステネス
の篩，繰り返し自乗法，剰余の世界での四則演算

コードチャレンジ：基本課題#2-a [1点]

${}_nC_k$の998244353で割った余りを出力するコードを書いてください.

math.pow,　math.factorial,　その他mathの関数や組み込み関数，同様のライブラリ・関数等を<u>**使用せずに**</u>書いてください.

コードチャレンジ：基本課題#2-b [1.5点]

与えられた整数の範囲（L，R）において，Nも(N+1)/2も素数となるような奇数の総数を求めてください．

コードチャレンジ：基本課題#2-b [1.5点]

D - 2017-like Number Editorial

Time Limit: 2 sec / Memory Limit: 256 MB

配点 : 400 点

問題文

「N も $(N+1) \div 2$ も素数」を満たす奇数 N を **2017に似た数** とします。

Q 個のクエリが与えられます。

クエリ $i(1 \leqq i \leqq Q)$ では奇数 l_i, r_i が与えられるので、$l_i \leqq x \leqq r_i$ かつ **2017に似た数** となる奇数 x の個数を求めてください。

制約

- $1 \leqq Q \leqq 10^5$
- $1 \leqq l_i \leqq r_i \leqq 10^5$
- l_i, r_i は奇数
- 入力は全て整数

https://atcoder.jp/contests/abc084/tasks/abc084_d

コードチャレンジ：基本課題#2-b [1.5点]

考え方のヒント

素数の数え上げをしたい．→何を使う？

ただし，クエリが何個も飛んでくるので，毎回毎回
数え上げをすると遅い．

→1からiまでで該当する奇数の総数を予め計算しておき，それを利用することを考える．

コードチャレンジ：Extra課題#2 [3点]

累積和的な考え方を取り入れる整数関連の問題.