

هو العليم



مبانی کامپیوتر و برنامه‌سازی

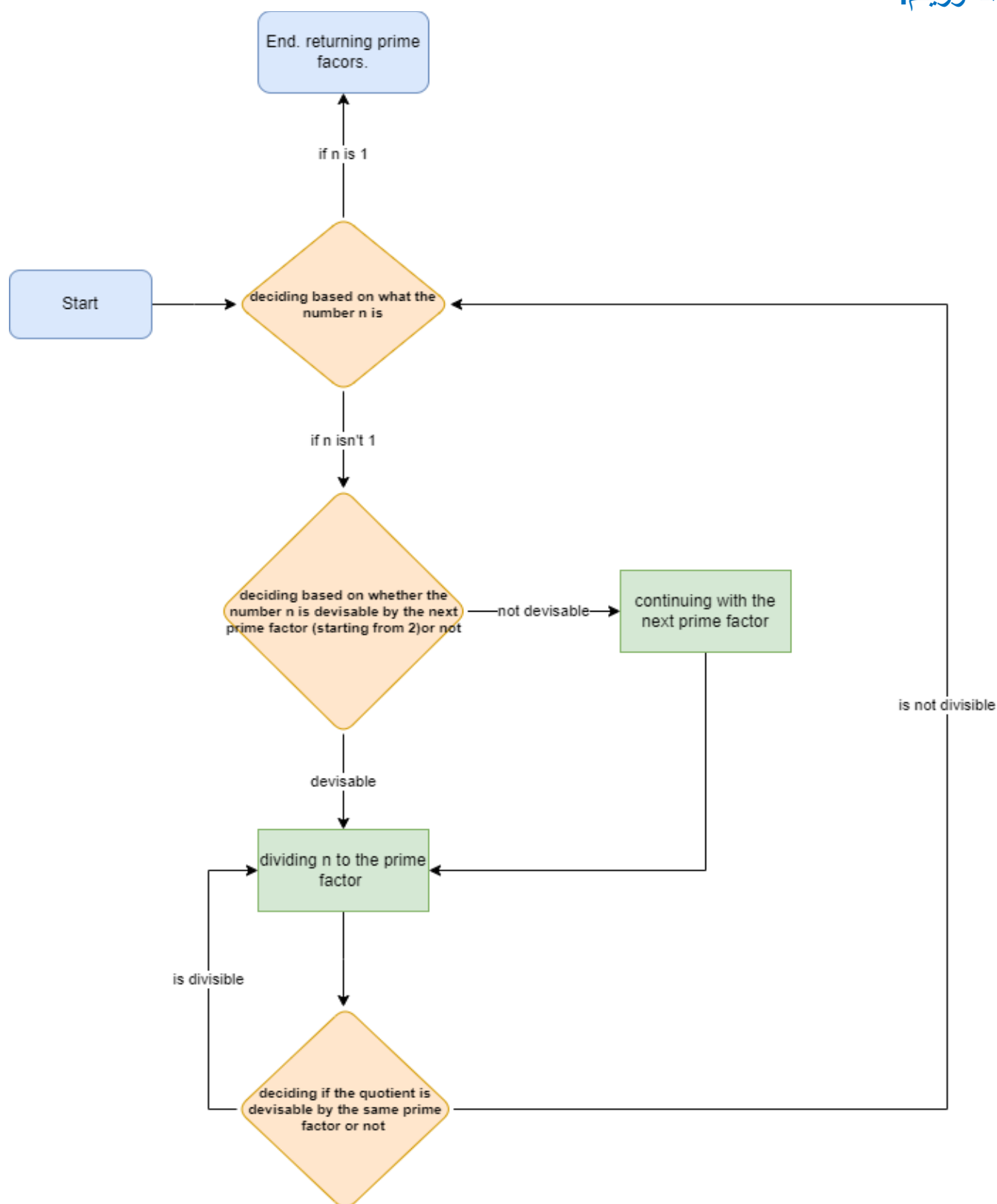
نیمسال اول سال تحصیلی 1403 – 1402

مسائل برنامه‌نویسی

پوریا مرادپور - 4024023040

مسأله 1: تجزیه اعداد صحیح به عامل‌های اول آنها

الگوریتم:



برنامه:

is_prime(number: int) -> bool:

با تقسیم همه اعداد طبیعی کوچکتر مساوی نصف تقسیم صحیح بر دو بر عدد اول بودن آن عدد را مشخص میکند

prime_number_generator(start_point: int = 1):

با شروع از ارگومان start_point و استفاده از تابع is_prime عدد اول بعدی را برمیگرداند

prime_factorize(number: int, prime_factor: int = 2, prime_factors=None):

تابع بازگشتی که با تقسیم عدد های اول بر عدد ورودی یک دیکشنری شامل عامل های اول به عنوان کلید و توان آن ها به عنوان مقدار برمیگرداند نحوه کار دقیق تر این تابع در بخش الگوریتم توضیح داده شده است.

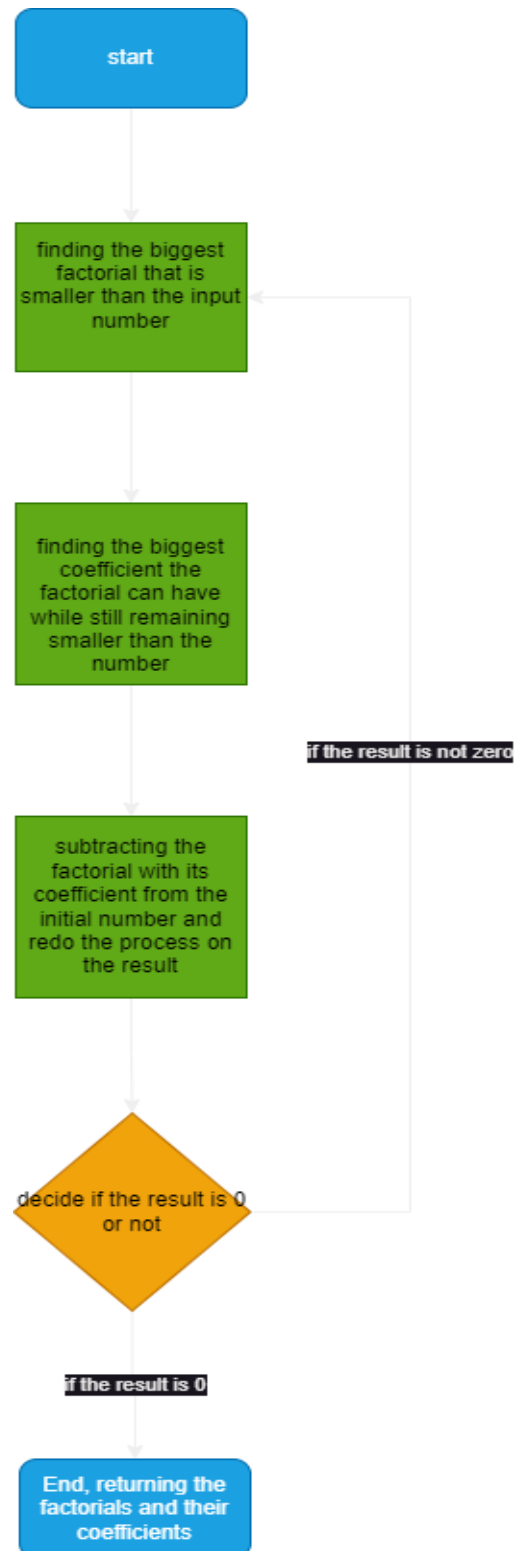
interface():

برای ارتباط با کاربر و هندل کردن ارور های احتمالی

مراجع:
از منبع خاصی استفاده نشد

مسألة 2: بسط فاكتريلي اعداد طبيعي

الـكـورـيـتم:



برنامه:

factorial(number: int) -> int:

محاسبه فاکتوریل عدد با هر بار کم کردن از آن و ضرب کردن تن از تابع بازگشتی استفاده نشد چون با توجه به recursion limit پایتون برای اعداد بالای 994 جواب نمیداد

biggest_factorial(number: int) -> int:

با استفاده از تابع factorial بزرگترین فاکتوریلی که همچنان کوچکتر از عدد است را محاسبه میکند (با شروع از عدد و یکی یکی کم کردن از آن)

factorial_coefficient(number: int, fctl: int) -> int:

با شروع از ضریب 1 و یکی یکی بزرگ کردن آن بزرگترین ضریب فاکتوریل یک عدد که حاصل آن همچنان از عدد کمتر است را محاسبه میکند

factorial_expansion(number: int) -> dict:

با استفاده از توابع قبلی و هر بار کم کردن حاصل فاکتوریل از عدد و ضریبش و دوباره تکرار کردن این پروسه برای حاصل تفریق بسط فاکتوریل عددی را به صورت یک دیکشنری که کلیدهای آن اعدادی هستند (که فاکتوریل آن ها مد نظر است) و مقادیر آن ها ضرایبشان است برمیگرداند

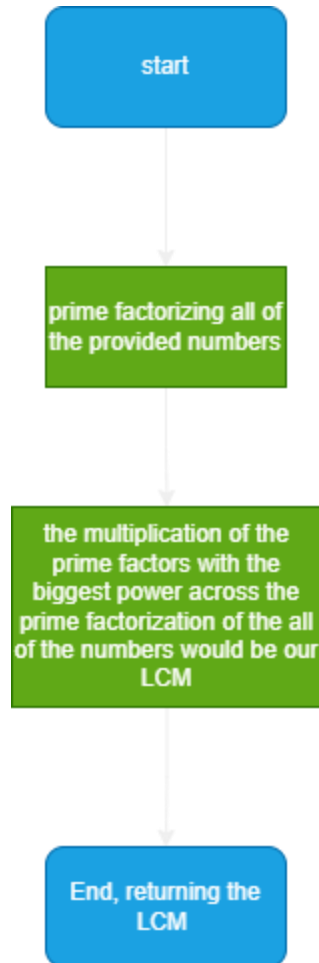
interface () :

برای ارتباط با کاربر و هندل کردن ارور های احتمالی

مراجع:
از منبع خاصی استفاده نشد

مسأله 3: محاسبه کوچک‌ترین مضرب مشترک چند عدد

الگوریتم:



برنامه:

`lcm(numbers: list) -> int:`

با استفاده از تابع `prime_factorize` سوال 1، همه اعداد وارد شده را به عوامل اول تجزیه میکند سپس همه عوامل موجود در تجزیه ها را به صورت کلید در یک دیکشنری ذخیره میکند. در نهایت بزرگترین توان هر توابعاملن را به عنوان مقدار آن کلید قرار میدهد در این حالت ضرب کلید های دیکشنری که هر کدام به توان مقدار خود رسیده اند مقدار کم م را به ما میدهد. جزییات بیشتر در کامنت های کد موجود هستند.

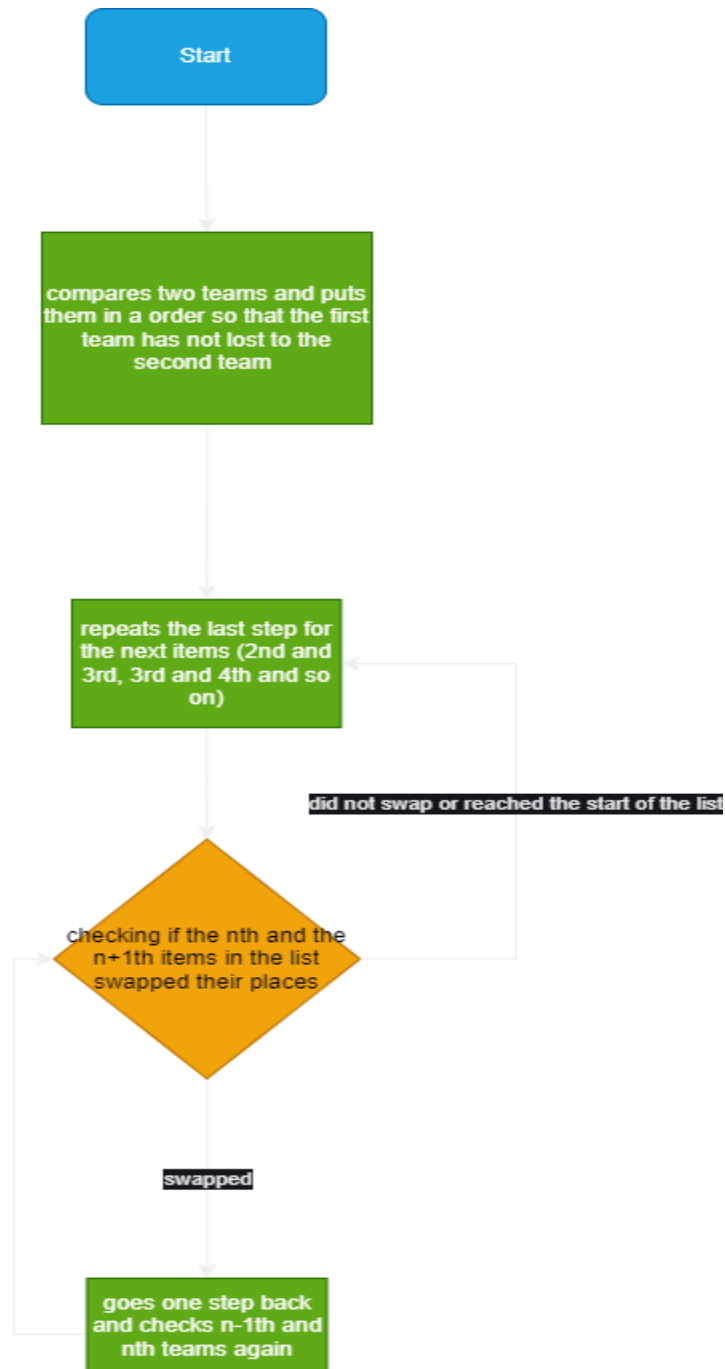
Interface () :

برای ارتباط با کاربر و هندل کردن ارور های احتمالی , ورودی گرفتن از کاربر را تا جایی ادامه میدهد که کاربر عدد 0 را وارد کند

مراجع:
از منبع خاصی استفاده نشد

مسأله 4: مرتب سازی تیم ها

الگوریتم:



برنامه:

bubble_sort(lst: list):

لیست را به روش بابل سورت به صورت صعودی مرتب میکند.

team_list(teams_quantity: int) -> list:

با گرفتن تعداد تیم ها لیستی از تیم ها با شماره های 1 تا n ایجاد میکند.

compare(team_1: int, team_2: int, games_data: dict, teams_list: list) -> bool:

با ورودی گرفتن دیکشنری که در آن نتایج بازی مشخص شده (ساختار دیکشنری در کامنت های خط 21 تا 27 کد توضیح داده شده است) مکان دو تیمی که در ارگومان ها مشخص شده اند را در لیست تیم ها به طوری مرتب میکند که تیم قبلی از تیم بعدی شکست نخورده باشد. مقدار بازگشتی آن یک Boolean است که در صورت تغییر جای دو تیم False و در غیر اینصورت True است

organize(games_data: dict, teams_list: list) -> list:

لیست تیم ها را دوتا دوتا پیمایش میکند و با استفاده از تابع compare آن را مرتب میکند اما تغییر جای دو تیم ممکن است ترتیب تیم های قبلی را هم از بین ببرد به همین دلیل بعد از هر تغییر دو تیم قبلی هم با تابع compare بازرسی میشوند و در صورت جابجایی دوباره این کار تا جایی ادامه پیدا میکند که به دو تیم ابتدایی لیست برسیم.

Interface () :

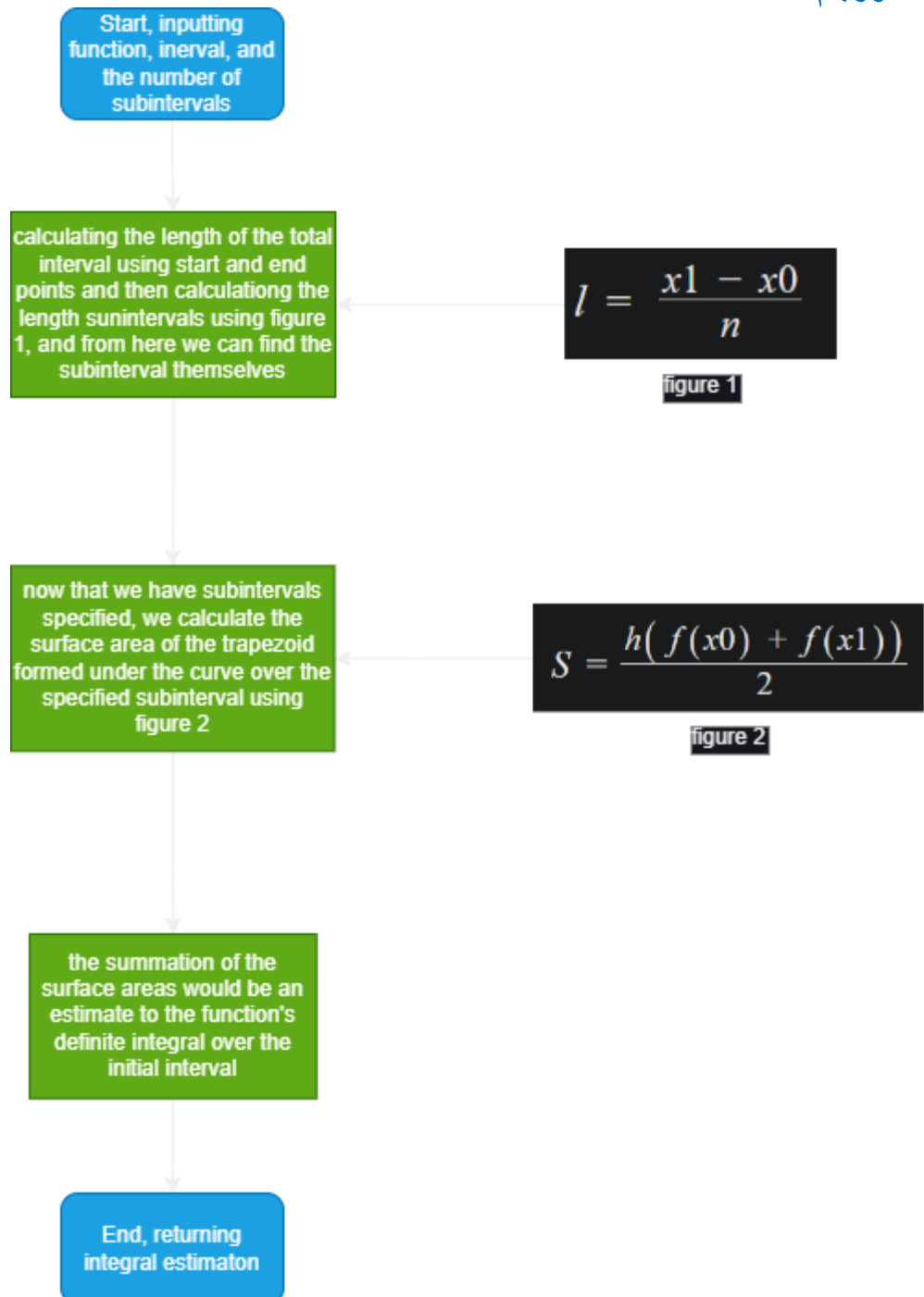
برای ارتباط با کاربر و هندل کردن ارور های احتمالی. تعداد تیم ها را از کاربر ورودی میگرد و با استفاده از تابع subset_generator (مربوط به سوال 7) زیرمجموعه های دو عضوی تیم ها را که معادل بازی ها می باشند تولید کرده و سپس نتایج بازی ها را از کاربر دریافت کرده و در نهایت با این اطلاعات دیکشنری لازم برای استفاده در تابع organize را میسازد.

مراجع:

از منبع خاصی استفاده نشد

مسأله 5: انتگرال گیری عددی

الگوریتم:



برنامه:

power(x: int | float, n: int) -> float:

با استفاده از ضرب توان را محاسبه میکند.

absolute_value(x):

قدر مطلق عدد را برمیگرداند.

Square_root(x: int | float) -> float:

با استفاده از روش هرون ریشه دوم مثبت را محاسبه میکند به این صورت که با یک حدس اولیه که خود عدد است شروع میکند و با استفاده از رابطه هرون حدس را به ریشه دوم نزدیک میکند تا جایی که توان دوم حدس با عدد اولیه کمتر از 0.0001 اختلاف داشته باشد و در این صورت حدس را برمیگرداند.

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{S}{x_n} \right)$$

sin(x: int | float) -> float:

با استفاده از بسط مکلورن برای سینوس آن را تا $n=10$ محاسبه میکند.

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

class Polynomial

کلاس برای چند جمله ای هاست که برای تعریف یک شی باید درجه چندجمله ای و ضریب هر درجه را در لیستی که به عنوان آرگومان گرفته میشود از درجه بالا به پایین مشخص کرد سپس با استفاده از متد `find_y` و ورودی دادن یک عدد متناظر آن را محاسبه میکند

trapezoidal_rule(interval: tuple, func) -> float:

با توجه به تابع و بازه داده شده و با استفاده از رابطه مساحت ذوزنقه مقدار تقریبی مساحت تابع و محور x ها را مشخص میکند

$$S = \frac{h(f(x_0) + f(x_1))}{2}$$

```
estimate_integral(interval: tuple,  
sub_intervals_quantity: int, func) -> float:
```

بازه را به تعداد مشخص شده ای زیربازه تقسیم میکند و با استفاده از تابع `trapezoidal_rule` مساحت زیر هر یک را محاسبه میکند و حاصل جمع آن ها را برمیگرداند

Interface () :

برای ارتباط با کاربر و هندل کردن ارور های احتمالی.

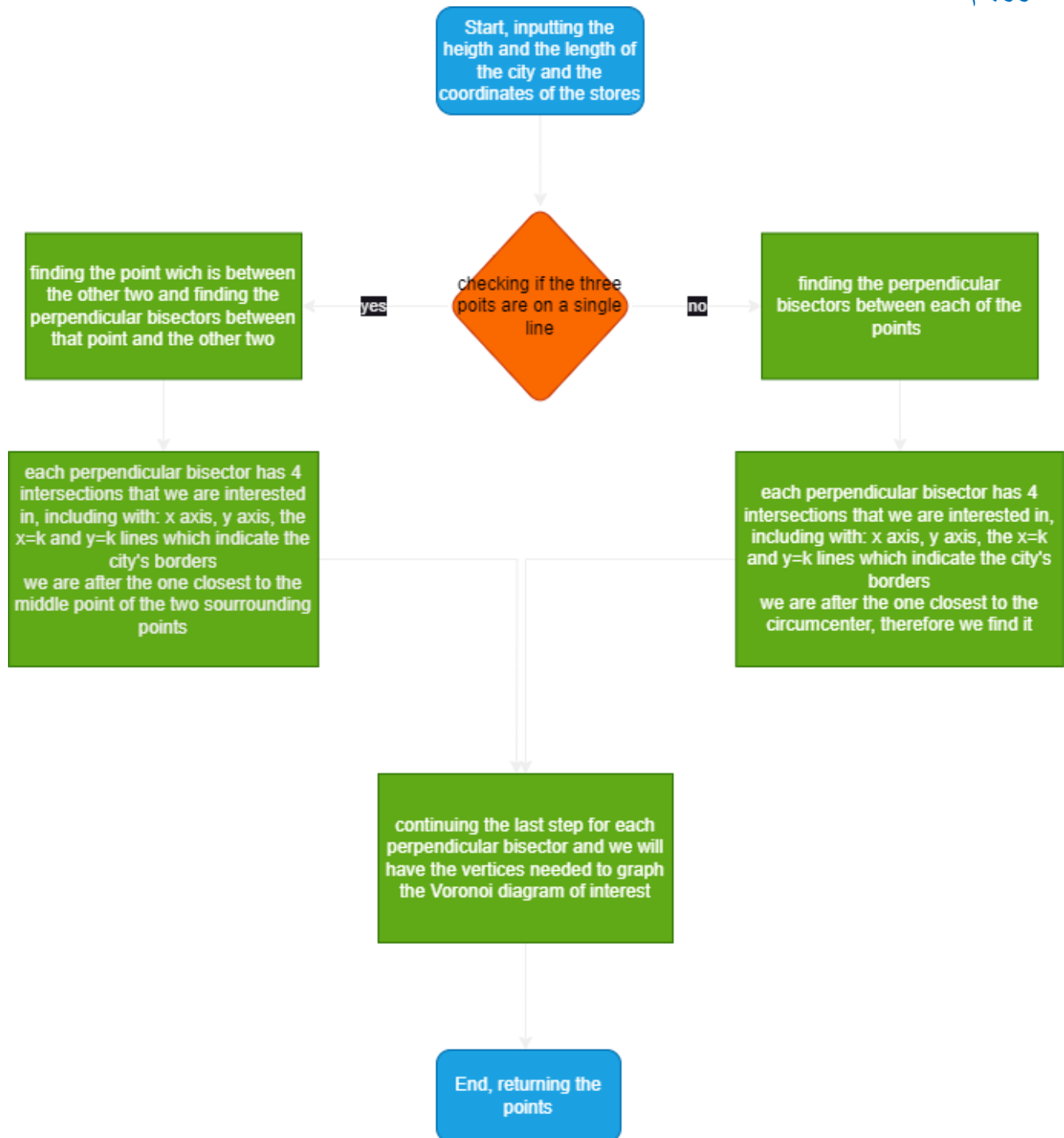
مراجع:

[منبع روش هرون برای محاسبه ریشه دوم](#)

[منبع بسط مکلورن برای محاسبه سینوس](#)

مسألة 6: تجزیه شهر

الگوریتم:



برنامه:

```
line_intersection(line_1: tuple[float, float],  
line_2: tuple[float, float]) -> tuple[float,  
float]:
```

با استفاده از تابع `equation_solver` (مربوط به سوال 10) محل برخورد دو خط را محاسبه میکند

`perpendicular_bisector_finder`

معادله عمود منصف بین دو نقطه را محاسبه میکند (برای حالات خاصی که به صورت $x=k$ و $y=k$ هم جواب میدهد) و به صورت تاپل (slope, constant) برمیگرداند (جزئیات به صورت کامنت در کد موجودند)

```
line_drawer(point_1: tuple[float, float], point_2:  
tuple[float, float]) -> tuple[float | None, float]:
```

بین دو نقطه یک خط رسم میکند (برای حالت خاص $y=k$ هم جواب میدهد)

```
on_a_line(point_1: tuple[float, float], point_2:  
tuple[float, float], point_3: tuple[float, float])  
-> bool:
```

با استفاده از تابع `line_drawer` خطی بین دو نقطه رسم میکند اگر نقطه سوم نیز روی آن بود یعنی هر سه روی یک خط هستند

`middle_point_finder`

اگر هر سه نقطه روی یک خط باشند حتما یکی از آن ها میان دو نقطه دیگر است و هر دو مولفه آن نیز بین مولفه های نقاط دیگر است، این تابع با استفاده از سورت کردن آن نقطه را پیدا میکند

`voronoi_diagram_outlines`

سه نقطه فروشگاه هارا ورودی میگیرد و در صورتی که نقاط روی یک خط باشند عمود منصف های نقطه میانی با دو نقطه دیگر را برمیگرداند و در غیر اینصورت سه عمود منصف موجود بین هر دو نقطه را برمیگرداند

```
euclidean_dist(center: tuple[float, float],  
*points: tuple) -> dict:
```

فاصله اقلیدسی دو نقطه را بدست می آورد

key_of_smallest_value(dictionary: dict) :

یک دیکشنری را به عنوان ورودی میگیرد و کلید کوچکترین مقدار آن را برمیگرداند که در تابع voronoi_diagram_vertex_finder لازم است

voronoi_diagram_vertex_finder

هر عمود منصف با چهار خط برخورد دارد: محور x محور y و خط های $y=k$ و $x=k$ که به صورت مرز های شهر هستند. ما نزدیک ترین برخورد به محل همرسی عمود منصف ها (در حالتی که روی یک خط نباشند) و یا به نقطه وسط دو نقطه کناری (در صورتی که روی یک خط باشند) نیاز داریم این تابع این نقاط را پیدا کرده و برمیگرداند (جزئیات بیشتر در کامنت های این تابع موجودند)

Interface () :

برای ارتباط با کاربر و هندل کردن ارور های احتمالی.

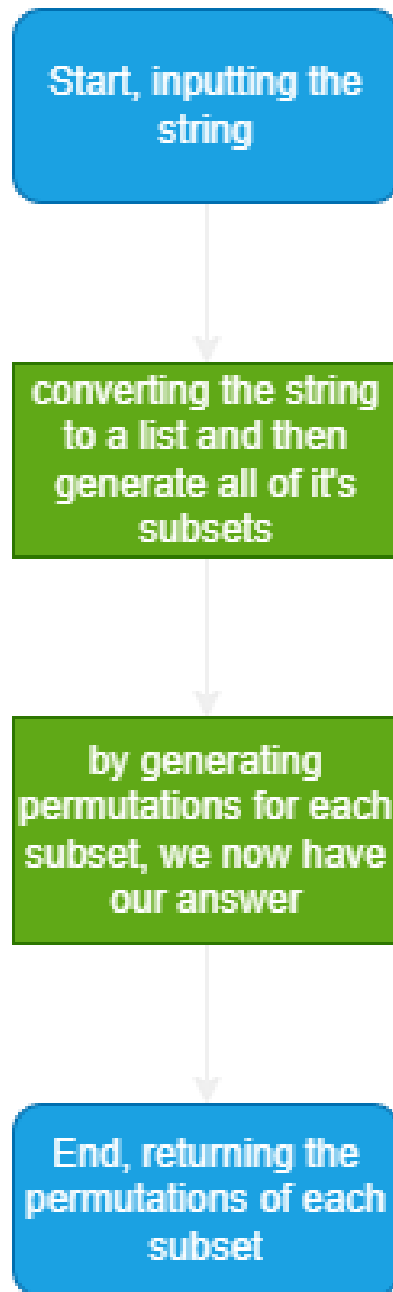
مراجع:

[Introduction to Voronoi diagrams](#)

[The Fascinating World of Voronoi Diagrams](#)

مسأله 7: ساخت رشته‌ها

الگوریتم:



برنامه:

permutation_generator(lst: list, start: int, end: int, permutations: list) -> list:

یک تابع بازگشتی که جایگشت های لیست ورودی را محاسبه میکند و بر میگرداند

plain_dict(initial_set: list) -> dict:

یک لیست ورودی میگیرد و دیکشنری ای با کلید های اعداد یک تا طول لیست و مقادیر لیست خالی میسازد. این دیکشنری برای تابع subset_generator لازم است.

subset_generator(initial_set: list, subsets: dict) -> dict:

زیرمجموعه های لیست را حساب میکند و در دیکشنری به تفکیک تعداد اعضای زیرمجموعه ذخیره میکند

subset_permutation(initial_set: list) -> list:

جایگشت های زیرمجموعه های لیست را برمیگرداند که جواب سوال است.

Interface () :

برای ارتباط با کاربر و هندل کردن ارور های احتمالی , ورودی گرفتن از کاربر را تا جایی ادامه میدهد که کاربر عدد 0 را وارد کند

مراجع:

منبع الگوریتم تولید جایگشت ها (تسلطی به زبان استفاده شده در این مقاله ندارم و صرفا از الگوریتم استفاده کردم)

مسأله 8: ارزیابی عبارت‌های پسوندی

الگوریتم:



برنامه:

```
operate(left_num: float, right_num: float,  
operator: str) -> float:
```

عملیات را با عملگر مشخص شده روی اعداد مشخص شده اجرا میکند

```
prefix_expression_calculator(expression: str) ->  
float:
```

عبارت پسوندی را برعکس میکند و برای هر کارکتر [ک میکند که عملگر است یا عملوند. اگر عملوند بود آن را به لیستی اضافه میکند و اگر عملگر بود با آن عملیاتی بین دو عملوند آخر لیست اجرا میکند و نتیجه آن را جایگزین این دو عملوند در لیست میکند و این کار را تا پایان عبارت پسوندی ادامه میدهد

Interface () :

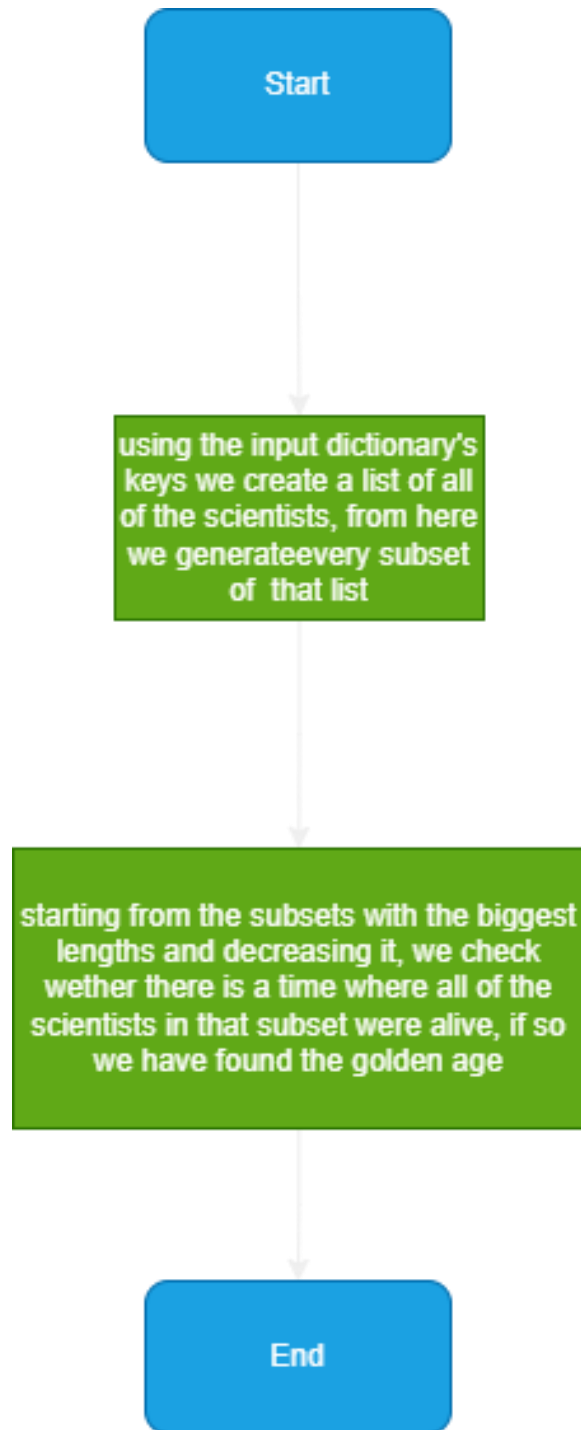
برای ارتباط با کاربر و هندل کردن ارور های احتمالی , ورودی گرفتن از کاربر را تا جایی ادامه میدهد که کاربر عدد 0 را وارد کند.

مراجع:

از منبع خاصی استفاده نشد

مسأله 9: دوره زمانی طلایی

الگوریتم:



برنامه:

intersect(first_interval: tuple, second_interval: tuple) -> tuple | None:

اشتراک دو بازه را با پیدا کردن نقطه شروع بازه بزرگتر و نقطه پایان بازه کوچکتر محاسبه میکند و برمیگرداند

multi_intersection(intervals: list) -> tuple | None:

با استفاده از خاصیت $A \cap B \cap C = (A \cap B) \cap C$ و تابع intersect اشتراک n بازه را محاسبه میکند

reverse_bubble_sort(lst: list):

سورت به روش بابل سورت اما لیست به صورتی نزولی برگردانده میشود (در تابع golden_age_calc لازم است)

descend_keys(dctnr: dict) -> dict:

دیکشنری داده شده را طوری مرتب میکند که کلید هایش نزولی باشند که برای iterate کردن دیکشنری در تابع golden_age_calc لازم است

golden_age_calc(world_science_history: dict) -> tuple:

با استفاده از توابع قبلی و تابع subset_generator تمام زیرمجموعه های دانشمندان را محاسبه میکند. سپس از زیرمجموعه هایی با بیشترین تعداد عضو شروع میکند و به دنبال زمان مشترکی در زندگی تمام دانشمندان آن زیرمجموعه میگردد هر جا این زمان پیدا شد دوران طلایی را پیدا کرده ایم

Interface () :

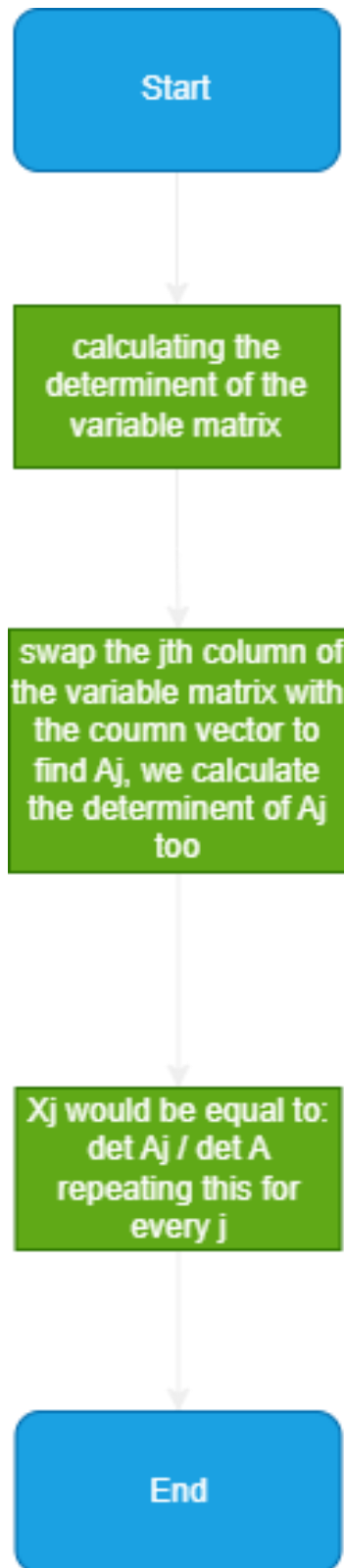
برای ارتباط با کاربر و هندل کردن ارور های احتمالی. ورودی گرفتن از کاربر را تا جایی ادامه میدهد که کاربر عدد 0 را وارد کند.

مراجع:

از منبع خاصی استفاده نشد

مسأله 10: حل دستگاه‌های معادلات خطی

الگوریتم:



برنامه:

is_n_n(matrix: list) -> bool:

چک میکند که ماتریس مربعی است یا خیر

sub_matrix_generator(matrix: list, row: int, column: int) -> list:

یک زیر ماتریس را با حذف سطر و ستون مشخص شده ایجاد میکند

determinant(matrix: list) -> int:

تابعی بازگشتی که با استفاده از رابطه موجود در پی دی اف سوالات دترمینان ماتریس را محاسبه میکند (جزئیات بیشتر در کامنت های کد موجود است)

column_swapper(initial_matrix: list, new_column: list, column_index: int) -> list:

یک ستون را با یک ستون خارجی جابجا میکند که برای اجرای قاعده کرامر لازم است

cramer_rule(variable_matrix: list, column_vector: list, x_index: int) -> float:

با استفاده از روش کرامر A_j را محاسبه کرده و تقسیم بر ماتریس ضرایب میکند تا X_j محاسبه شود

equation_solver(variable_matrix: list, column_vector: list) -> dict:

روش کرامر را برای همه X ها اجرا میکند و نتایج را به صورت یک دیکشنری برمیگرداند

Interface () :

برای ارتباط با کاربر و هندل کردن ارور های احتمالی.

مراجع:

از منبع خاصی استفاده نشد