تمرینات فصل چهار

|  |  |  |
| --- | --- | --- |
| نام: امیرحسام بهمن‌خواه | نام: مریم رضائی | نام: محمدحسین فرخی |

**1.** مسأله‌ی کوله­پشتی را در نظر بگیرید: عنصر با وزن­های معلوم و ارزش­های معلوم و یک کوله­پشتی با ظرفیت ، داده شده­اند؛ باارزش­ترین زیرمجموعه­ای از این عناصر را پیدا کنید که بتوان آنها را درون کوله­پشتی جا داد.

**الف)** الگوریتم جستجوی کامل برای مسأله کوله­پشتی را با شبه­کد توصیف کنید.

**ب)** کارایی زمانی الگوریتم را با نماد مجانبی بیان کنید.

**پ)** برنامه­ای برای پیاده­سازی الگوریتم بنویسید. و بزرگ­ترین مقداری از را که به ازای آن، برنامه، روی رایانه شما، در کمتر از 1 دقیقه قادر به تولید خروجی باشد، پیدا کنید.

**جواب:**

**1 – الف)** با استفاده از روشی شبیه به جستجوی سطری در ماتریس و به کارگیری یک ماتریس کمکی، ابتدا زیرمجموعه‌ی مورد نظر از پر ارزش‌ترین عناصر را پیدا می‌کنیم و سپس با پیمایش در ماتریس مقادیر مجموعه را یافته و در یک لیست قرار می‌دهیم تا به عنوان خروجی برگردانیم.

کارکرد الگوریتتم به این شکل است که اگر ابعاد *i* و *j* ماتریس کمکیِ خالی را به ترتیب تعداد عناصر و ظرفیت کوله‌پشتی در نظر بگیریم، هر خانه دارای مجموع ارزش‌های پر ارزش‌ترین عناصری است که index آن‌ها کوچکتر مساوی *i ِ* خانه باشد و جمع وزنشان کوچکتر مساوی *j* ِ آن. با تشکیل این ماتریس و سپس حرکت از آخرین خانه‌ی آن به بالا تا جایی که جمع ارزش‌ها تغییر کند (یعنی عنصرِ هم index با *i* ِ خانه در ارزش تغییری داده و در کوله‌پشتی قرار داده شده است) و سپس به چپ با شرط قبل، می‌توانیم عناصر مورد نظر را بیابیم و در یک آرایه خروجی دهیم. شبه کد الگوریتم این بخش در صفحه‌ی بعد به طور کامل قابل مشاهده است.

**1 – ب)** از آنجا که الگوریتم از سه بخش مستقل تشکیل شده است که دو بخش اول دو حلقه‌ی تودرتو با تکرار و بخش سوم یک حلقه با تکرار است، کارایی زمانی الگوریتم بر اساس جفت حلقه‌ی تودرتوی اول محاسبه می‌شود (که برای ساخت ماتریس کمکی‌ست) و برابر با O() می‌باشد.

**Algorithm: Backpack(*capacity*, *n*, *weights*[*w*1, …, *wn*], *values*[*v*1, …, *vn*])**

// finds most valuable subset whose sum of weights does not exceed the capacity

// Input: list *weights* of *n* weights, list *values* of *n* values, the *capacity*, *n*

// Output: subset of most valuable elements

*matrix* ← [[0 **for** *w* ← 0 **to** *capacity*] **for** *i* ← 0 **to** *n*]

*SelectedItems* ← list()

**for** *i* ← 0 **to** *n* **do**

**for** *w* ← 0 **to** *capacity* **do**

**if** *i* = 0 **or** *w* = 0 **then**

*matrix*[*i*][*w*] ← 0

**elif** *weights*[*i* – 1] ≤ *w* **then** // weight of item *i* – 1

*previous* ← *matrix*[*i* – 1][*w*]

*new ← values*[*i* – 1] + *matrix*[*i* – 1][*w* – *weights*[*i* – 1]]

*matrix*[*i*][*w*] ← max(*previous*, *new*)

**else then**

*matrix*[*i*][*w*] ← *matrix*[*i* – 1][*w*]

*totval* ← *matrix*[*n*][*capacity*] // max possible amount for total value

*w* ← *capacity*

**for** *i* ← *n* **to** 0 **do**

**if** *totval* < 1 **then**

**break** // reached end of backtracking in matrix

**if** *totval* = *matrix*[*i* – 1][*w*] **then**

**continue** // total value in this row equal to total value in above index

**else then** // if above value in column (same capacities) differs

*SelectedItems*.append(*weights*[*i* – 1]) // current item is included in pack

*totval* ← *totval* – *values*[*i* – 1]

*w* ← *w* – *weights*[*i* – 1]

**return** *SelectedItems*

**1 – پ)** با کمک ماژول‌های time و random در پایتون، لیست‌های تصادفی با طول‌های متفاوت برای وزن‌ها و ارزش‌ها درست شدند و با در نظر گرفتن ظرفیت به اندازه‌ی میانگین وزن‌ها، برنامه چندبار و با بازه‌ی مختلف *n* اجرا شد تا بازه‌ای که زمان اجرایش در حدود یک دقیقه بود یافت و انتخاب شود. در آخر اعداد صحیح درون این بازه (که از *n* = 540 تا 560 بود) تک تک امتحان شدند که نتایج آن بر حسب ثانیه در زیر آورده شده و کد برنامه (و مشخصات کامپیوتر) در پیوست 1 ذکر شده است.

|  |  |
| --- | --- |
| **Length (n)** | **Time (s)** |
| 540 | 58.446 |
| 541 | 58.564 |
| 542 | 59.713 |
| 543 | 60.815 |
| 544 | 60.33 |
| 545 | 67.632 |
| 546 | 66.601 |
| 547 | 60.744 |
| 548 | 59.996 |
| 549 | 60.501 |
| 550 | 61.417 |
| 551 | 60.308 |
| 552 | 61.76 |
| 553 | 61.35 |
| 554 | 62.38 |
| 555 | 63.126 |
| 556 | 63.612 |
| 557 | 63.315 |
| 558 | 68.38 |
| 559 | 64.296 |
| 560 | 66.92 |

2. نسخه غیربازگشتی (پایین – ­به - ­بالا) الگوریتم مرتب­سازی درجی را در نظر بگیرید:

**الف)** الگوریتم مرتب­سازی درجی را به شکل بازگشتی (بالا- به­- پایین) بنویسید. و آنگاه با تشکیل رابطه­های بازگشتی و حل آنها، کارایی زمانی الگوریتم را در بهترین حالت و در بدترین حالت تعیین کنید. کارایی فضایی الگوریتم چقدر است؟ از نظر مصرف حافظه، کدام یک از دو الگوریتم بازگشتی یا غیربازگشتی کاراتر است؟

**ب)** از آنجا که نسخه غیربازگشتی الگوریتم، در زمان اجرا از پشته استفاده نمی­کند، انتظار ما این است که در عمل، الگوریتم غیربازگشتی، اندکی سریع­تر باشد از الگوریتم بازگشتی. درستی این ادعا را با نوشتن و اجرای برنامه­های پیاده­سازِ دو الگوریتم غیربازگشتی و بازگشتی (روی ورودی­های مختلف) تحقیق کنید.

**پ)** نسخه غیربازگشتی الگوریتم مرتب­سازی درجی را به گونه­ای تغییر دهید که در هر تکرار خود، برای یافتن مکان درج یک عدد در زیرآرایه مرتب، از جستجوی دودویی استفاده کند (نه از جستجوی خطی). آیا با این تغییر، به الگوریتمی خواهیم رسید که کارایی آن در بدترین حالت باشد؟ در پاسخ به این پرسش، کارایی زمانی الگوریتم حاصل را اندازه بگیرید.

**جواب:**

**2 – الف)** در بازگشتی کردن الگوریتم مرتب‌‌سازی درجی، با یک فراخوانی در هر مرحله، *n* – 1 عنصر اول آرایه را مرتب می‌کنیم و سپس عنصر *n*ام را با عناصر قبل مقایسه کرده و در جای خود قرار می‌دهیم. برای رابطه‌ی بازگشتی آن، در هر سری فراخوانی یک بازگشت برای یکی کمتر از پارامتر داشته و یک حلقه‌ی جدا داریم که تعداد تکرار آن به مقدار پارامتر ورودی‌ست. اگر آرایه مرتب باشد هیچگاه وارد حلقه نشده و کارایی O(*n*) خواهد بود. در بهترین حالت، آرایه یک عضو دارد یعنی پس کارایی O(1) می‌باشد. اما در بدترین حالت آرایه عکس حالت مرتب چیده شده است پس تمامی حلقه‌ها را وارد می‌شود. یعنی:

برای کارایی فضایی الگوریتم غیر بازگشتی مرتب‌سازی درجی مشاهده می‌کنیم که هیچ متغیر بزرگ اضافه‌ای علاوه بر آرایه‌ی ورودی تشکیل نمی‌شود پس کارایی فضایی آن O(1) است، درحالی که برای الگوریتم بازگشتی، در هر فراخوانی داده‌ها در پشته‌ی فراخوانی ذخیره می‌شوند که طول آن در نهایت به طول اصلی آرایه‌ی ورودی خواهد بود؛ بنابراین کارایی فضایی O(*n*) می‌باشد.

**Algorithm: RecursiveInsertionSort(*A*[0…*n* – 1])**

// implements insertion sort algorithm recursively

// Input: array *A*

// Output: sorts array *A*

**if** *n* ≤ 1 **then**

**return** // ends recursion

RecursiveInsertionSort(*A*[:*n* – 1])

*temp* ← *A*[*n* – 1]

*pick* ← *n* – 2

**while** *pick* ≥ 0 **and** *A*[*pick*] > *temp* **then**

*A*[*pick* + 1] ← *A*[*pick*]

*pick* ← *pick* – 1

*A*[*pick* + 1] ← *temp*

**2 – ب)** کد پایتون برنامه‌های نوشته شده برای اندازه‌گیری زمان اجرای الگوریتم بازگشتی و غیربازگشتی مرتب‌سازی درجی در پیوست 2 قرار داده شده است. داده‌های استخراج شده از فایل CSV خروجی در جدول و نمودار در زیر بر حسب میلی‌ثانیه نمایش داده شده‌اند. مشاهده می‌کنیم زمان اجرای الگوریتم بازگشتی به جز چند حالت استثنا، به طور کلی کمی از الگوریتم غیربازگشتی بیشتر است.

|  |  |  |
| --- | --- | --- |
| **Length** | **Non-R (ms)** | **R (ms)** |
| 100 | 0.79 | 1.008 |
| 150 | 2.003 | 2.063 |
| 200 | 2.681 | 3.325 |
| 250 | 5.273 | 5.025 |
| 300 | 7.082 | 7.679 |
| 350 | 9.333 | 10.647 |
| 400 | 16.241 | 13.508 |
| 450 | 16.01 | 16.745 |
| 500 | 21.829 | 21.611 |
| 550 | 26.911 | 24.504 |
| 600 | 30.897 | 31.873 |
| 650 | 40.572 | 36.862 |
| 700 | 43.087 | 40.638 |
| 750 | 47.128 | 47.246 |
| 800 | 51.849 | 54.302 |
| 850 | 61.632 | 66.907 |
| 900 | 74.969 | 72.446 |
| 950 | 74.958 | 79.688 |
| 1000 | 83.337 | 85.432 |

**2 – پ)** الگوریتم جستجوی دودویی، خود یک الگوریتم بازگشتی‌ست و می‌توانیم به راحتی با نوشتن جداگانه‌ی آن و فراخوانی در میان الگوریتم مرتب سازی درجی، جایگاه مورد نظر برای قرار دادن عنصر را بیابیم و سپس آن را در موقعیت قرار دهیم. اما جالب‌تر است که از آنجا که مرتب‌سازی درجی‌مان غیربازگشتی‌ست، جستجو را هم باز کرده و به صورت یک حلقه درون خود الگوریتم مرتب‌سازی تعبیه کنیم. در این حلقه با مقایسه‌ی عنصر با بقیه‌ی عناصر آرایه و همچنین تعیین اینکه قبل یا بعد آن‌ها باید قرار گیرد، جایگاه مورد نظر پیدا می‌شود.

**Algorithm: BinaryInsertionSort(*A*[0…*n* – 1])**

// implements insertion sort algorithm with non-recursive binary search

// Input: array *A*

// Output: sorts array *A*

**for** *pick* ← 0 **to** *n* **do**

*value* ← *A*[*pick*]

*left* ← 0

*right* ← *n* – 1

**while** *left* < *right* **do**

*mid* ← int(*left*/2 + *right*/2)

**if** *A*[*left*] = *value* **then**

*temp* ← *left*

**if** *A*[*right*] = *value* **then**

*temp* ← *right*

**if** *A*[*mid*] = *value* **then**

*temp* ← *mid*

**if** *A*[*mid*] > *value* **then**

*right* ← *mid*

**else then**

*left* ← *mid*

*A* ← *A*[:*temp*] + [*value*] + *A*[*temp*:*pick*] + *A*[*pick*+1:]

برای کارایی زمانی الگوریتم حاصل مشاهده می‌کنیم که حلقه‌ی for بیرونی حتماً *n* بار تکرار را دارد و حلقه‌ی while درونی که جستجوی دودویی‌ست، تنها log *n* تکرار دارد؛ اما با این وجود نمی‌توانیم سریع نتیجه بگیریم که کارایی زمانی الگوریتم وابسته به حاصل ضرب این دو حلقه‌ی تودرتوست، زیرا در خط آخر درون حلقه‌ی for عنصر مورد نظر را از جای خود برداشته و با هل دادن عناصر بزرگتر از آن به راست، آن را در مکان مورد نظر قرار می‌دهیم. واضح است که در بدترین حالت که آرایه برعکس چیده شده باشد، این عنصر همواره به اول آرایه رفته و همه‌ی عناصر را به راست هل می‌دهد؛ پس هر بار *n* جابه‌جایی انجام داده و کارایی همواره O(*n*2) است.

تنها در حالتی که نوع داده به شکلی باشد که قیمت مقایسه از جابه‌جایی بیشتر باشد، الگوریتم دارای جستجوی دودویی از خطی کاراتر خواهد بود زیرا تعداد مقایسه‌های آن در بدترین حالت خطی–لگاریتمی‌ست.

**3.** الگوریتم اقلیدس، الگوریتم تقلیل­وحلی است که بزرگ­ترین مقسوم علیه مشترک دو عدد صحیح و را مبتنی بر رابطه بازگشتی محاسبه می­کند.

**الف)** درستی رابطه بازگشتی

را ثابت کنید.

**ب)** مبتنی بر این رابطه بازگشتی، یک الگوریتم تقلیل­وحل دیگر برای محاسبه ب.م.م دو عدد صحیح بنویسید.

**پ)** با این فرض که اعداد و ، آنقدر بزرگ باشند که نتوان عملیات حسابی روی آنها را در زمان به انجام رساند، کارایی زمانی الگوریتم اقلیدس و کارایی زمانی الگوریتمی را که نوشته­اید، اندازه بگیرید. کدام یک از این دو الگوریتم تقلیل­وحل، کاراتر است؟

**جواب:**

**3 – الف)** برای اثبات درستی رابطه‌ی بازگشتی ارائه شده نیاز است صحت هر شرط را بر اساس قوانین ریاضی نشان دهیم تا ببینیم که با هر تغییر، مقدار ب.م.م. حفظ شده و تساوی برقرار است. اگر :

در شرط اول برای زمانی که هر دو عدد داده شده زوج باشند، بنابر تعریف ب.م.م. می‌دانیم که عدد 2 حتماً در مقدار ب.م.م. آن‌ها خواهد بود، پس می‌توانیم آن 2 را خارج کرده و اعداد را بر 2 تقسیم کنیم تا این مقسوم‌الیه مشترکشان حذف شود. اما در حالتی که تنها یکی از اعداد زوج باشد (شرط دوم)، بر اساس ب.م.م. داریم که 2 در ب.م.م. آن‌ها نخواهد بود، پس عدد زوج را تا جای ممکن با رابطه‌ی بازگشتی بر 2 تقسیم می‌کنیم تا زوج بودن آن بر طرف شود. در آخر تنها حالتی می‌ماند که هر دو عدد فرد باشند و برای اثبات درستی محاسبه‌ی ارائه شده برای ب.م.م. در این شرط داریم که اگر بنابراین و . همچنین با خواص عاد کردن داریم . از طرفی می‌دانیم و هر دو فرد هستند پس تفریق آن‌ها زوج خواهد بود و از آنجا که مقدار ب.م.م. دو عدد فرد بود و زوج نیست، می‌توانیم تفریق را بر 2 تقسیم کنیم و عدد حاصل را کوچکتر کنیم پس و بنابراین .

این رابطه بازگشتی تا جایی که ممکن است ادامه پیدا می‌کند، یعنی مقادیر چپی عدد کوچکتر تا جایی از عدد بزرگتر کم می‌شوند که عدد بزرگتر، بیت‌های قابل توجه خود را (که به تعداد تفاوت طول بیتی دو عدد هستند) از دست داده و نتواند به چپ جابه‌جا شود.

**3 – ب)** با تعمیم رابطه‌ی داده شده و اضافه کردن شروطی برای مقایسه‌ی اعداد در هر بار فراخوانی بازگشتی تابع و تعیین عدد بزرگتر و همچنین شرط صفر بودن، الگوریتم زیر را می‌نویسیم.

**Algorithm:** **BinaryGCD(*m*, *n*)**

// implements and generalizes the binary division to calculate GCD

// Input: two numbers *m* and *n*

// Output: the greatest common divisor of *m* and *n*

**if** |*n*| > |*m*| **then**

**return** BinaryGCD(*n*, *m*)

**elif***n* = 0 **then**

**return** *m*

**elif** *m* **and** *n* **are even then** // can check their remainder in division by two with %

**return** 2 × BinaryGCD(*m*/2, *n*/2)

**elif** *m* **is** **even and** *n* **is odd then**

**return** BinaryGCD(*m*/2, *n*)

**elif** *n* **is** **even and** *m* **is odd then**

**return** BinaryGCD(*m*, *n*/2)

**else then**

**return** BinaryGCD((|*m*| – |*n*|)/2, *n*)

**3 – پ)** در تحلیل کارایی الگوریتم اقلیدس اصلی ، بنابر اثبات ارائه شده توسط گابریل لامه تعداد مراحل اجرا شده توسط الگوریتم با دنباله‌ی فیبوناچی ارتباط دارد به طوری که اگر تعداد این تکرارها *n* باشد، کوچکترین اعدادی که این امر برایشان صدق کند اعدادی در دنباله‌ی فیبوناچی هستند که اگر عدد کوچکتر ورودی *b* باشد، آنگاه و که هر کدام بزرگترین اعدادی در دنباله‌ی فیبوناچی هستند که از ورودی‌ها کوچکتر یا مساوی‌اند. بنابراین د کارایی زمانی آن برای *n* تکرار به کمک نسبت طلایی φ داریم:

از طرفی برای الگوریتم داده شده (که الگوریتم دودویی ب.م.م. و الگوریتم استاین نیز خوانده می‌شود) از آنجا که تنها از تقسیم بر دو استفاده می‌شود و در کامپیوتر اعداد به طور دودویی شناخته شده و با آن‌ها کار می‌شود، قیمت خود تقسیم کمتر است. اگر پارامتر را بر اساس تعداد بیت‌های عدد بزرگتر بگیریم که این تعداد بیت‌ها می‌باشد، تعداد مراحل تفریق و جابه‌جایی به راستِ اجرا شده توسط الگوریتم در بدترین حالت با آن برابر بوده و الگوریتمی خطی داریم. اما چون که اعداد بزرگ هستند و خود اعمال تفریق و جابه‌جایی بیتی O(*m*) زمان می‌برند، کارایی زمانی الگوریتم O(*m*2) است.

مشاهده می‌کنیم که کارایی زمانی هر دو الگوریتمِ ب.م.م. مربعی بوده و تفاوت زیادی ندارند. با این وجود قابل به ذکر است که الگوریتم دودویی در هر حال تعداد اعمال بیتی کمتری را نسبت به الگوریتم اصلی انجام می‌دهد و مدلی بهینه شده از آن است، این در حالی‌ست که کارایی زمانی الگوریتم اقلیدس مقداری از الگوریتم استاین کمتر است زیرا *h* لگاریتم عدد کوچکتر در مبنای ده است و *m* لگاریتم عدد بزرگتر در دو و می‌دانیم همواره پس برای مقادیر بزرگ، توان دوی این اعداد از هم دورتر می‌شوند.

**4.** الگوریتم درج در درخت دودویی جستجو را در نظر بگیرید:

**الف)** الگوریتم درج را به شکل بازگشتی بنویسید. و آنگاه با تشکیل یک رابطه­ بازگشتی و حل آن در حالات خاص، کارایی زمانی الگوریتم در بهترین حالت و در بدترین حالت را تعیین کنید. کارایی فضایی الگوریتم را نیز تعیین کنید. از نظر مصرف حافظه، کدام یک از دو الگوریتم بازگشتی یا غیربازگشتی کاراتر است؟

**ب)** یک درخت دودویی جستجوی گره­ای را می­توان با درج یک به یک عناصر در درخت ابتدائاً تهی ساخت. با تحلیل­های دقیق ریاضی نشان دهید که کارایی زمانی الگوریتم ساخت درخت، در بدترین حالت و در بهترین حالت خواهد بود.

**پ)** یکی از مواردی که بدترین حالت کارایی زمانی الگوریتم ساخت درخت، رخ خواهد داد، وقتی خواهد بود که مقدار همه‌ی عناصری که قرار است در درخت درج شوند، یکسان باشند. اما با ترفندهایی می­توان کارایی الگوریتم را در چنین مواردی بهتر کرد:

* یک ترفند این است که الگوریتم به گونه­ای تغییر داده شود که در هنگام درج یک عنصر، اگر کلید آن با کلید یک گره برابر باشد، متناوباً به سمت چپ و سمت راست آن گره حرکت کند. مثلاً اگر الگوریتم بخواهد تعدادی عنصر را درخت درج کند و مقدار چهار عنصر از آن عناصر 5 باشد، بعد از آنکه اولین 5 را در درخت درج کرد، آنگاه برای درج دومین 5 ، وقتی در طول مسیر حرکت خود به گره­ای برسد که مقدار 5 در آن ذخیره شده باشد، مسیر سمت چپ (راست) گره را انتخاب می­کند؛ و برای درج سومین 5 ، وقتی در طول مسیر حرکت خود به گره­ای برسد که مقدار آن 5 باشد، مسیر سمت راست (چپ) گره را انتخاب می­کند؛ و برای درج چهارمین مقدار 5 ، وقتی در طول مسیر حرکت خود به گره­ای برسد که مقدار آن 5 باشد، باز مسیر سمت چپ (راست) گره را انتخاب می­کند.
* ترفند دیگر این است که الگوریتم به گونه­ای تغییر داده شود که اگر در هنگام درج یک عنصر، کلید آن با کلید یک گره برابر باشد، گره­ای جدید ایجاد می­کند و کلید را در آن ذخیره می­کند و آن را به گره موجود اضافه می­کند (اگر چنین لیستی از گره­های حاوی مقادیر یکسان، قبلاً ایجاد شده باشد، الگوریتم صرفاً گره­ای جدید به لیست اضافه می­کند و کلید را در آن ذخیره می­کند).

اگر رفتار الگوریتم درج، برای درج کلیدهای تکراری در یک درخت، طبق ترفند اول تغییر داده شود، کارایی بدترین حالت الگوریتم ساخت درخت، برای درج عنصر با کلیدهای مساوی چقدر خواهد بود؟ طبق ترفند دوم چطور؟

**جواب:**

**4 – الف)** برای بازگشتی کردن الگوریتم کافی‌ست در هر بار فراخوانی بازگشتی، فرزند چپ یا راست گره‌ای که روی آن هستیم را بسته به بزرگتر یا کوچکتر بودن مقدار آن از عنصر مورد نظر برای درج، به عنوان ریشه‌ی یک زیردرخت به تابع داده و عملیات بازگشتی را تا جایی ادامه دهیم که گره خالی باشد یا برابر با عنصر باشد.

**Algorithm: RecursiveTreeInsert(*Tree*, *root*, *x*)**

// implements tree insertion algorithm recursively

// Input: *root* of *Tree* and element *x* to insert into tree

// Output: modifies *Tree* to contain *x* as leaf

**if** *root*.value = *None* **then**

*root* ← *x*

**return**  // stops recursion

**else then**

**if** *root*.value < *x* **then**

RecursiveTreeInsert(*Tree*, *root*.right, *x*)

**else then**

RecursiveTreeInsert(*Tree*, *root*.left, *x*)

در این الگوریتم برای هر بار فراخوانی الگوریتم، با فرض اینکه مقایسه و جایگذاری عنصر کاراییِ ثابت دارد، یک عمل ثابت و یک فراخوانی بازگشتی تابع برای نصف درخت یا به زبان دیگر، زیردرختی از درخت داده شده با نیمی از عناصر آن داریم. همچنین در بهترین حالت فراخوانی بازگشتی انجام نشده و کارایی ثابت است، پس:

البته این محاسبات برای ارتفاع در یک درخت است که ارتفاع معمول یک درخت دودویی متعادل و کمترین ارتفاع ممکن برای یک درخت دودویی کج می‌باشد. از آنجایی که بیشترین مقدار ممکن برای ارتفاع یک درخت دودویی است، الگوریتم در بدترین حالت تا انتهای بلندترین شاخه‌ی درخت رفته و گره‌ها را بررسی می‌کند، پس کارایی آن O(*n*) خواهد بود.

از طرفی با بررسی الگوریتم غیر بازگشتی مشاهده می‌کنیم که متغیر بزرگی اضافه بر درخت تشکیل نمی‌شود، پس کارایی فضایی آن O(1) است؛ در حالی که در الگوریتم بازگشتی با هر فراخوانی یک پشته شکل می‌گیرد که به علت پیش رفتن یک‌طرفه‌ی الگوریتم به اندازه ارتفاع درخت، کارایی فضایی آن را O(*h*) یعنی همان O(log *n*) می‌کند. بنابراین الگوریتم غیر بازگشتی از نظر حافظه نسبت به بازگشتی کاراتر است.

**4 – ب)** در ساخت یک درخت دودویی با درج عناصر به کمک الگوریتم بالا، در بهترین حالت ترتیبِ دادن عناصر به الگوریتم به گونه‌ای‌ست که درخت متعادل شده و ارتفاع آن کمترین حالت ممکن یعنی شود، زیرا در این صورت تعداد گره‌های visit شده برای قرار دادن عنصر بنابر محاسبات ریاضی بخش الف خواهد بود. این تحلیل را می‌توان به شکل زیر نیز نوشت:

از آن‌جا که عملیات برای عنصر انجام می‌شود، کارایی زمانی الگوریتم در بهترین حالت O(*n* log *n*) است. اما در بدترین حالت، عناصر به ترتیب از کوچک به بزرگ به الگوریتم داده می‌شوند و به همین علت از آنجا که هربار کلید گره‌ها را مقایسه می‌کند و بر اساس بزرگ و کوچکی‌شان به چپ و راست می‌رود، همواره درخت تشکیل شده کج و نامتعادل خواهد بود (یا عناصر تکراری‌اند و همه در یک طرف قرار می‌گیرند) به طوری که ارتفاع آن بیشترین حالت ممکن و برابر با *n* – 1 به دست می‌آید. پس برای *n* عنصر هر کدام *n* – 1 مقایسه داشته و کارایی در بدترین حالت O(*n*2) می‌باشد.

**4 – پ)** در ترفند اول، از آنجا که عناصر تکراری همه در یک طرف جمع نشده و ارتفاع به شکل متعادل‌تری افزایش میابد (عملاً مانند حالت عادی هر بار تکرار حلقه بر دو تقسیم می‌شود) پس ارتفاع حدوداً سقف پایینی log *n* بوده و کارایی زمانی آن بنا بر بخش الف، در بهترین حالت O(*n* log *n*) خواهد بود (مگر اینکه عوامل دیگری باعث کجی درخت شوند). در ترفند دوم ارتفاع حتی کمتر خواهد بود زیرا گره‌ای جداگانه به عنوان فرزند ایجاد نشده و به ارتفاع اضافه نمی‌شود. پس تعداد مقایسه‌ها کمتر می‌شود اما کارایی زمانی همان O(*n* log *n*) باقی می‌ماند که بهترین حالت ممکن است (مگر اینکه باز عواملی دیگر باعث کجی شوند و کارایی O(*n*2)شود).

**5.** یکی از مسائل الگوریتمی که در عمل زیاد با آن مواجه می­شویم، مسأله پیدا کردن همه عناصری در یک مجموعه است که مقدار آنها برابر یا بین دو مقدار خاص باشد. برای مثال:

* کارمند آموزش یک دانشگاه بزرگ ممکن است ملزم به ایجاد لیستی از دانشجویان دانشگاه باشد که معدل آنها برابر یا بین دو مقدار خاص باشد؛
* یک شهروند ممکن است بخواهد در یک فروشگاه اینترنتی، کالاهایی (مثلاً کتاب­هایی) را بیابد که قیمت آنها برابر یا بین دو مقدار خاص باشد؛
* بازرس یک سازمان دولتی بزرگ ممکن است نیازمند تعیین افرادی در سازمان باشد که میزان حقوق آنها برابر یا بین دو مقدار خاص باشد.

این مسائل در واقع نوعی مسأله جستجو هستند؛ بیان کلی و دقیق این نوع مسأله جستجو این است: مجموعه شامل عنصر و دو مقدار و داده شده است. همه عناصری در مجموعه را که مقدار آنها برابر یا بین دو مقدار و باشد، بیابید.

طراحی یک الگوریتم جستجوی کارا برای حل چنین مسأله­ای، مستلزم این است که داده­ها به شکلی مناسب در حافظه چیده شده باشند.

**الف)** با این فرض که عناصر مجموعه ، عدد باشند و به ترتیبی نامعلوم در یک آرایه ذخیره شده باشند، الگوریتمی برای حل مسأله طراحی کنید. کارایی زمانی الگوریتمی که به آن رسیده­اید، چقدر است؟

**ب)** اکنونبا این فرض که عناصر مجموعه ، عدد باشند و به ترتیب صعودی (نانزولی) در یک آرایه ذخیره شده باشند. الگوریتمی کارا برای حل مسأله طراحی کنید. کارایی زمانی الگوریتمی که به آن رسیده­اید، چقدر است؟

**پ)** آرایه مرتب گرچه مفید است اما در وضعیتی که علاوه بر جستجو در مجموعه ، بخواهیم عنصری را به مجموعه اضافه کنیم یا عنصری را از مجموعه حذف کنیم، گزینه مناسبی برای چینش داده­ها در حافظه نیست. در چنین وضعیتی بهتر این است که عناصر به شکل یک درخت دودویی جستجو (د.د.ج) در حافظه چیده شوند. پس با این فرض که عناصر مجموعه ، عدد باشند و در گره­های یک د.د.ج ذخیره شده باشند، الگوریتمی کارا برای حل مسأله طراحی کنید. کارایی زمانی الگوریتمی که به آن رسیده­اید، چقدر است؟

**جواب:**

**5 – الف)** از آنجایی که مجموعه‌ی *S* نامرتب است و ما تمام عناصری را می‌خواهیم که در بازه‌ی داده شده قرار دارند، نیاز است که حتماً تک تک عناصر را بررسی کنیم. با استفاده از یک حلقه برای پیش رفتن خطی در مجموعه‌ی ورودی، کارایی زمانی این الگوریتم O(*n*) است که *n* تعداد اعضای مجموعه است.

**Algorithm: Non-SortedArraySearchInterval(*S*[0…*n* – 1], *k*1, *k*2)**

// finds all set elements in given interval

// Input: set *S*, lower bound *k*1, upper bound *k*2

// Output: list of elements between upper and lower bounds

*verified* ← list()

**for** *elements* **in** *S* **do**

**if** *element* ≥ *k*1 **and** *element* ≤ *k*2 **then**

*verified.*append(*element*)

**return** *verified*

**5 – ب)** حال که مجموعه مرتب است می‌توانیم با تغییر در الگوریتم جستجوی دودویی، index حدهای پایین و بالای بازه در آرایه (و یا در صورت وجود نداشتن خود مقادیر، عناصری که به آن‌ها نزدیک تر هستند) را بیابیم که کارایی این بخش O(log *n*) خواهد بود. اگر قصد ما یافتن تعداد عناصر درون بازه بود، می‌توانستیم با کم کردن مقادیر indexها، پاسخ را در کارایی مرتبه‌ی لگاریتمی پیدا کنیم. اما از آنجا که می‌خواهیم خود عناصر را بازگردانیم و طبیعت صورت سوال از ما‌ می‌خواهد که تمام عناصر بازه را بازگردانیم، نیاز است حتماً در مجموعه پیش رفته و عناصر میان این دو index را خروجی دهیم که در بدترین حالت این به معنای بررسی کردن حداقل یک بار تمامی عناصر مجموعه است. پس کارایی زمانی الگوریتم برابر با O(*n*) می‌باشد.

راه دیگری برای حل مسئله این است که مانند بخش الف در مجموعه پیش رفته و عناصر را با حد بسته‌ی بالا و پایین مقایسه کنیم و موارد یافت شده را در آرایه‌ای ذخیره کنیم. از آنجا که این حل بسیار ساده بوده و کارایی زمانی آن همچنان O(*n*) است در حالی که تغییر و استفاده از جستجوی دودویی راهکار جالب‌تری می‌باشد، در صفحه‌ی بعد، شبه کد الگوریتم با استفاده از جستجوی دودویی نوشته شده است.

**Algorithm: SortedArraySearchInterval(*S*[0…*n* – 1], *k*1, *k*2)**

// finds all sorted set elements in given interval

// Input: set *S*, lower bound *k*1, upper bound *k*2

// Output: list of elements between upper and lower bounds

**if** *S*[0] > *k*2 **or** *S*[*n* – 1] < *k*1 **then**

**return** “out of range”

**if** *S*[0] ≥ *k*1 **then**

*lower* ← 0

**elif** *S*[0] < *k*1 **then**

*low* ← 0

*high* ← *n* – 1

**while** *low* ≤ *high* **do**

*mid* ← int((*high* + *low*)/2)

**if** *S*[*mid*] ≥ *k*1 **then**

*high* ← *mid* – 1

**else then**

*low* ← *mid* + 1

*lower* ← *low*

**if** *S*[*n* – 1] ≤ *k*2 **then**

*upper* ← *n* – 1

**elif** *S*[*n* – 1] > *k*2 **then**

*low* ← 0

*high* ← *n* – 1

**while** *low* ≤ *high* **do**

*mid* ← int((*high* + *low*)/2)

**if** *S*[*mid*] ≤ *k*2 **then**

*low* ← *mid* + 1

**else then**

*high* ← *mid* – 1

*upper* ← *high*

*verified* ← *S*[*lower*:*upper*]

**return** *verified*

**5 – پ)** برای جستجو در یک د.د.ج. دارای داده‌های مسئله، در الگوریتم بازگشتی جستجوی درخت به شکلی تغییر می‌دهیم که کلید هر گره‌ها با حدهای بازه مقایسه کند و درصورت موجود بودن آن در بازه، آن گره را بازگرداند و الگوریتم را به شکل بازگشتی برای دو فرزند چپ و راست گره اجرا کند تا جایی که به انتهای درخت و یا پایان بازه برسد.

شبه کد زیر به شکل تابع سازنده پایتون نوشته شده است اما می‌توان مقادیر را در پشته‌ای قرار داد و در آخر آن را به عنوان خروجی بازگرداند. این پشته می‌تواند به عنوان یک ویژگی برای درخت تعریف شده باشد یا می‌تواند به عنوان ورودی انتخابی پنجم در هر بار فراخوانی بازگشتی به تابع ورودی داده شود و در آخر بازگردانده شود.

**Algorithm: BinaryTreeSearchInterval(*Tree*, *root*, *k*1, *k*2)**

// yields all set elements in given interval using binary search tree recursively

// Input: *Tree* and the *root* to search from, lower bound *k*1, upper bound *k*2

// Output: elements between upper and lower bounds (yields as generator)

**if** *root*.value = *None* **or** *root*.value < *k*1 **or** *root*.value > *k*2 **then**

**return** // stops recursion

**elif** *k*1 ≤ *root*.value ≤ *k*2 **then**

**yield** *root*

BinaryTreeSearchInterval(*Tree, root*.left, *k*1, *k*2)

BinaryTreeSearchInterval(*Tree, root*.right, *k*1, *k*2)

برای تحلیل کارایی زمانی الگوریتم بالا رابطه‌ی بازگشتی آن را می‌نویسیم. در هر مرحله، دو فراخوانی بازگشتی برای زیر درخت (با نصف گره‌های درخت قبل) داشته و یک عملیات ثابت داریم، پس می‌نویسیم:

از آنجایی که در بخش ب توضیح دادیم که کارایی زمانی الگوریتم نمی‌تواند کمتر از خطی باشد، می‌توانیم با الگوریتم ساده اندیشانه و خطی DFS بر روی درخت نیز به شکل زیر، به جستجو در آن بپردازیم.

**Algorithm: TreeDFSSearchInterval(*Tree*, *k*1, *k*2)**

// finds all set elements in given interval using binary search tree

// Input: *Tree*, lower bound *k*1, upper bound *k*2

// Output: list of elements between upper and lower bounds

*verified* ← list()

*S* ← stack()

*S*.push(*Tree*.root)

**while** *S* **is not** *Empty* **do**

*node* ← *S*.pop()

**if** *node* **is not** *visited* **then**

**mark** *node* **as** *visited*

**if** *k*1 ≤ *node* ≤ *k*2 **then**

*verified*.append(*node*)

**for all edges from** *node* **to** *neighbor* **in** *Tree*.Edges(*node*) **do**

*S*.push(*neighbor*)

**return** *verified*

**6.** شما در نقش تحلیلگر داده­ها، می­خواهید از دو پایگاه داده مجزا، داده­هایی را (که بازیابی آنها وقتگیر است) بازیابی و تحلیل کنید. فرض کنید هر پایگاه شامل مقدار عددی است (یعنی در مجموع مقدار موجود است) و اینکه هیچ دو مقداری یکسان نیستند. تنها راهی که شما می­توانید به این مقادیر دسترسی داشته باشید، این است که از پایگاه داده­ها پرسش کنید؛ در هر بار پرسش، می­توانید مقدار را به یکی از دو پایگاه داده بدهید تا آن پایگاه داده، اُمین مقدار از نظر کوچکی را که شامل می­شود، برگرداند.

شما دوست دارید که میانه مجموعه مقداری را که طبق تعریف، اُمین مقدار مجموعه از نظر کوچکی است، تعیین کنید. از آنجا که پرسش­ها وقتگیر هستند، شما دوست دارید که میانه را با کمترین تعداد پرسش ممکن، تعیین کنید. الگوریتمی ارائه کنید که مقدار میانه را با حداکثر پرسش پیدا کند.

**جواب:** با مقایسه‌ی میانه‌های دو آرایه به طور بازگشتی و روش تقلیل و حل برای بررسی زیرمجموعه‌هایی از آرایه‌های اصلی می‌توانیم الگوریتم زیر را بنویسیم. این الگوریتم تا جایی پیش می‌رود که طول آرایه‌ها به یکی از شرط‌های اولیه ارائه شده برسد. از آنجایی که الگوریتم برای هر بار فراخوانی بازگشتی طول آرایه‌ها را نصف می‌کند، عملکردی شبیه به جستجوی دودویی دارد و کارایی آن در بدترین حالت O(log*n*) است.

**Algorithm: FindMedian(*A*[0…*n* – 1], *B*[0…*n* – 1])**

// finds median of two data sets with containing numbers with sorted indices

// Input: two data sets *A* and *B* with *n* elements

// Output: median of the union of both sets

**if** *n* = 0 **then**

**return** 0

**elif** *n* = 1 **then**

**return** (*A*[0] + *B*[0]) / 2

**elif** *n* = 2 **then**

**return** (max(*A*[0], *B*[0]) + min(*A*[1], *B*[1]) ) / 2

**else then**

**if** *n* **is even then** // *n* % 2 = 0

*medianA* ← (*A*[*n*/2] + *A*[*n*/2 – 1]) / 2

*medianB* ← (*B*[*n*/2] + *B*[*n*/2 – 1]) / 2

**if** *medianA* > *medianB* **then** // length becomes *n*/2 + 1

**return** FindMedian(*A*[:*n*/2 + 1], *B*[*n*/2 – 1:])

**else then**

**return** FindMedian(*A*[*n*/2 – 1:], *B*[:*n*/2 + 1])

**else then**

*medianA* ← *A*[int(*n*/2)] // division gives floor so we round it down

*medianB* ← *B*[int(*n*/2)]

**if** *medianA* > *medianB* **then**

**return** FindMedian(*A*[:int(*n*/2) + 1], *B*[int(*n*/2):])

**else then**

**return** FindMedian(*A*[int(*n*/2):], *B*[:int(*n*/2) + 1])

**7.** یک مالک قهوه فروشی­های زنجیره­ای، قهوه­های اعلا و گرانی را با نام­های زیبای ایتالیایی می­فروشد. از آنجا که بیشتر مشتریان او دانشجویان دانشگاه هستند و او متوجه شده­ است که خیابانی در یک شهر دانشگاهی بزرگ وجود دارد که در امتداد آن، خوابگاه دانشجویی وجود دارد اما قهوه فروشی در آن خیابان وجود ندارد، تصمیم گرفته است که یک قهوه فروشی جدید را در جایی از این خیابان دایر کند که طبق معیاری، فاصله از آن تا خوابگاه­های مختلف، بهینه باشد.

او از شما خواسته است که الگوریتمی برای حل مسأله­اش طراحی کنید. برای ساده کردن مسأله، خیابان را یک خط تصور می­کنیم و هر خوابگاه را یک نقطه (عددی حقیقی) روی این خط. به علاوه، ما برای هر ، مقدار را که تعداد افرادی است که در خوابگاه زندگی می­کنند، می­دانیم. با در اختیار داشتن این داده­ها، شما می­خواهید ای (مکان قهوه فروشی) را پیدا کنید که مقدار تابع فاصله زیر را کمینه کند.

با این مفروضات، یک الگوریتم کارا را برای پیدا کردن نقطه ای که با قرار گرفتن قهوه فروشی در آنجا، طبق تابع فوق، فاصله­­ آن از خوابگاه­ها، به حداقل خواهد رسید، طراحی و توصیف کنید. زمان اجرای الگوریتم شما چقدر است؟ **جواب:**

برای اینکه تابع فاصله‌ی داده شده کمینه کند، باید هرکدام از جملات سری کمترین مقدار ممکن باشند؛ یعنی ضرب جمعیت هر خوابگاه در فاصله‌ی آن تا محل خوابگاه کوچکترین حالت باشد. می‌دانیم که در محاسبه فواصل نسبی و نقاط روی یک محور، می‌توانیم مبدا را (که برای فاصله در تابع بالا نسبت به جایگاه کافی‌شاپ است) برای تمامی نقاط به نقطه‌ای دیگر (یعنی ابتدای خیابان) تغییر دهیم و آن را بر عدد ثابت جمع جمعیت‌ها تقسیم کنیم و همچنان نسبت و خواص (کمینه شدن) را حفظ کنیم. پس داریم:

بنابراین تابع فاصله، زمانی کمینه می‌کند که مقدار *x* از خوابگاه‌ها فاصله‌ای متناسب با جمعیت آن‌ها داشته باشد یعنی برابر با میانگین وزنی مقادیر باشد. برای نوشتن الگوریتمی که میانگین وزنی مقادیر نامرتب داده شده را بیابد، می‌توانیم با گرفتن داده‌ها به صورت آرایه‌ای از زوج مرتب‌ها (برای هر *di، pi* آن را نیز داریم)، به کمک روش بازگشتی داده‌ها را نصف کرده و در هر دو بخشِ کاهش یافته پیش رویم تا جایی که به دو خوابگاه برسیم و برای دو خوابگاه میانگین وزنی را بیابیم. آنگاه به عقب برگردیم و هر بار این میانگین یافت شده (و جمعیتی که برای آن تا به حال محاسبه شده) را به عنوان کاندیدی برای *x* در نظر گرفته و میان خود *x*ها میانگین وزنی به دست آوریم تا در آخر به *x* مورد نظر برسیم.

**Algorithm: RecursiveNon-SortedWeightedAverage(*Dorms*[(*d*1, *p*1), …, (*dn*, *pn*)])**

// calculates weighted average of non-sorted array recursively

// Input: array *Dorms* of ordered pairs of *di* and *pi*

// Output: location *x* and the general *population* it supports, as a pair

**if** *n* = 1 **then**

**return** *Dorms*[0]

**elif** *n* = 2 **then**

*numerator* ← *Dorms*[0][0] × *Dorms*[0][1] + *Dorms*[1][0] × *Dorms*[1][1]

*population* ← *Dorms*[0][1] + *Dorms*[1][1]

**return** [*numerator* **/** *population*, *population*]

**else then**

*half* ← int(*n* / 2)

*first* ← RecursiveNon–SortedWeightedAverage(*Dorms*[:*half*])

*second* **←** RecursiveNon–SortedWeightedAverage(*Dorms*[*half*:])

*numerator* ← *first*[0] × *first*[1] + *second*[0] × *second*[1]

*population* ← *first*[1] + *second*[1]

**return** [*numerator* **/** *population*, *population*]

از آنجا که الگوریتم حتماً تمامی خوابگاه‌ها را یک بار بررسی می‌کند، کارایی زمانی آن O(*n*) می‌باشد. محاسبات ریاضی تحلیل آن و رابطه‌ی بازگشتی‌اش دقیقاً به مانند پاسخ سوال 5 پ است که به طور مفصل ذکر شده است.

لازم به ذکر است که می‌توانستیم به سادگی در آرایه پیش رویم و مقادیر جمعیت را با هم جمع کرده و ضرب جمعیت در جایگاه خوابگاه‌ها را نیز جمع کنیم، سپس در یک تقسیم نهایی میانگین وزنی را با الگوریتمی خطی به دست آوریم؛ اما از آنجا که حل ارائه شده در ابتدا جالب تر بود، به آن پرداختیم.

**8.** پلیس شهر رایانستان، همه خیابان­های شهر را یک طرفه کرده است. با وجود این، شهردار رایانستان ادعا می‌کند که هنوز راهی برای رانندگی قانونی از هر تقاطعی در شهر به هر تقاطع دیگر آن، وجود دارد. چون مخالفان شهردار قانع نمی­شوند و شهر هم بسیار بزرگ است، برای تعیین درستی یا نادرستی ادعای شهردار، یک برنامه رایانه­ای لازم است. و از آنجا که انتخابات شورای شهر به زودی برگزار خواهد شد و شهردار خدمتگزار هم نگران از دست دادن فرصت عظیم خدمت به شهروندان است، به سراغ شما آمده است. شما می­دانید که با توجه به بزرگی شهر و زمان اندکی که برای شهردار موجود است، تنها باید به دنبال الگوریتمی خطی برای اثبات درستی ادعای شهر بود.

**الف)** این وضعیت را به شکل یک مسأله نظریه گراف، مدل کنید و آنگاه الگوریتمی برای حل آن ارائه کنید که زمان اجرای آن (برحسب اندازه ورودی­هایش) خطی باشد.

**ب)** شهردار باهوش رایانستان، به این هم فکر کرده است که در صورتی که با الگوریتم شما، نادرستی ادعای او مشخص شد، به فکر طرح کردن ادعای ضعیف­تری باشد: اینکه اگر شما از ساختمان شهرداری رانندگی را شروع کنید، خیابان‌های یک طرفه را بپیمایید و به هر جایی که ممکن بود برسید، باز همیشه راهی برای آنکه به طور قانونی رانندگی کنید تا به ساختمان شهرداری برگردید، خواهید داشت.

این ادعای ضعیف­تر شهردار را هم به شکل یک مسأله نظریه گراف، مدل کنید و باز الگوریتمی خطی ارائه کنید که با آن بتوان درستی یا نادرستی ادعا را تحقیق کرد. **جواب:**

**8 – الف)** در یک گراف جهت‌دار می‌توان به سادگی با کمک پیماش عمقی یا سطحی، از یک رأس دلخواه شروع کرد و سنجید که آیا گراف همبند قوی‌ست یا خیر. اگر شهر را گرافی در نظر گیریم که تقاطع‌های آن رئوس و خیابان‌های آن یال‌های جهت‌دار باشند، می‌توانیم با کمک پیماش عمقی الگوریتمی خطی بنویسیم که از رأسی شروع به پیمایش کرده و تعداد رأس‌هایی که می‌تواند به آن‌ها مسیری پیدا کند را با تعریف یک شمارنده بشمارد. در پایان پیماش انجام شده، اگر شمارنده برابر با تعداد رأس‌ها بود، گراف همبندی قوی دارد.

البته باید توجه کرد که شرط مقایسه‌ی شمارنده باید پس از اتمام پیمایش از یک نقطه‌ی شروع دلخواه بررسی شود و نباید نقطه‌ی شروع‌های متفاوت بررسی شوند زیرا آنگاه پیمایش عمقی همواره تمامی رئوس را بازدید خواهد کرد و دیگر همبندی قوی بررسی نخواهد شد. در صفحه‌ی بعد الگوریتم شرح داده شده با استفاده از پیمایش عمقی غیربازگشتی پیاده‌سازی شده است که به کمک یک پشته برای شبیه‌سازی پشته‌ی خودکار ایجاد شده در توابع بازگشتی، رئوس بازدید شده را ذخیره کرده و برای هرکدام حلقه را تکرار می‌کند. الگوریتم کارایی O(|*V*| + |*E*|) دارد زیرا هر رأس و یال درون لیست‌های مجاورت را حداکثر یک بار بازدید می‌کند.

**Algorithm: IsDirectedGraphStronglyConnectedDFS(*G*)**

// implements iterative DFS to check if directed graph is strongly connected

// Input: graph *G* of city

// Output: returns *False* if not strongly connected, else returns *True*

**for** *v* **in** *G* **do** // pick any vertex to start from

**mark** *v* **as visited**

*counter* ← 1

*S* ← stack(*v*) // let *S* be a stack and push *v* into it

**while** *S* **is not empty**

*v* ← *S*.pop()

**for** *edge*(*v*, *w*) **in** *G*.Edges(*v*) **do** // for all *w* adjacent to *v*

**if** *w* **is not visited then**

**mark** *w* **as visited**

*counter* ← *counter* + 1

*S*.push(*w*)

**if** *counter* ≠ length(*G*.Vertices) **then** **return** *False*

**else return** *True*

**8 – ب)** می‌دانیم که الگوریتم پیمایش عمقی بر روی گراف جهت‌دار در اصل درحال تشکیل یک درخت است؛ یعنی زمانی از یک رأس دوری به خودش وجود دارد که در یکی از نواده‌های خود نیز باشد. بنابراین می‌توانیم با استفاده از پیمایش عمقی از ساختمان شهرداری شروع کرده و مسیرهایی که می‌توان طی کرد را بازدید کنیم؛ اگر در جایی از این مسیرها باز به ساختمان شهرداری رسیدیم یعنی دور وجود داشته و ادعای شهردار درست است، در غیر این صورت غلط است.

شبه کد الگوریتم در صفحه‌ی بعد قرار داده شده‌است. در این الگوریتم، گراف شهر است و رأس ورودی ساختمان شهرداری که در صورت رسیدن به آن در رأس‌های بازدید نشده با پیماش، وجود دوره را تایید می‌کنیم. کارایی الگوریتم که از DFS بر گرافی با لیست‌های مجاورت استفاده می‌کند O(|*V*| + |*E*|) می‌باشد.

**Algorithm: DoesCycleExistDFS(*G*, *v*)**

// implements iterative DFS to check if there exists a cycle from given vertex

// Input: graph *G* of city and vertex *v* to check cycle for

// Output: returns *False* if no cycle exists, else returns *True*

*S* ← stack(*v*) // let *S* be a stack and push *v* into it

*start* ← *v* // store the beginning vertex

**while** *S* **is not empty**

*v* ← *S*.pop()

**for** *edge*(*v*, *w*) **in** *G*.Edges(*v*) **do** // for all *w* adjacent to *v*

**if** *w* **is not visited then**

**if** *w* = *start* **then** // beginning was never marked as visited

**return** *True*

**else mark** *w* **as visited**

*S*.push(*w*)

**return** *False*

**9.** آرایه شامل یک «عنصر اکثریت» است، اگر بیشتر از عناصر آن، یکسان باشند. الگوریتمی کارا طراحی کنید که آرایه­ای را به عنوان ورودی بگیرد و مشخص کند که آیا آرایه، یک عنصر اکثریت دارد یا خیر، و در صورت وجود عنصر اکثریت، آن را پیدا کند. در حالت کلی، عناصر آرایه، لزوماً از دامنه­های مرتب مانند اعداد صحیح نیستند و به همین دلیل، نمی­توان برای حل مسأله، از مقایسه­هایی به شکل «آیا است؟» استفاده کرد. (مثلاً فکر کنید که عناصر آرایه، فایل­های GIF باشند.) اما شما می­توانید به پرسش­هایی به شکل «آیا است؟» در زمان ثابت، پاسخ دهید.

**الف)** الگوریتمی ساده­اندیشانه برای این مسأله طراحی کنید. کارایی زمانی الگوریتم شما باید باشد و کارایی فضایی آن .

**ب)** الگوریتمی تقلیل­وحل برای این مسأله طراحی کنید. کارایی زمانی الگوریتم شما باید باشد و کارایی فضایی آن .

**پ)** اکنون الگوریتم تقلیل­وحلی طراحی کنید که با آن بتوان عنصر یا عناصری را که بیشتر از بار در آرایه تکرار شده باشند، پیدا کرد. ارایی زمانی الگوریتم شما باید باشد و کارایی فضایی آن .

**جواب:**

**9 – الف)** در الگوریتم ساده‌اندیشانه، با دو حلقه تمامی عناصر را مقایسه کرده و تکراری‌ها را می‌شماریم. سپس تعداد را با جزء صحیح نصف طول آرایه مقایسه می‌کنیم و در صورت بزرگ‌تر بودنش، آن عنصر را به عنوان عنصر مورد نظر ذخیره می‌کنیم. پر واضح است که آرایه نمی‌تواند بیشتر از یک عنصر اکثریت داشته باشد، زیرا تعداد آن باید از نصف تعداد کل بیشتر باشد. در آخر عنصر ذخیره شده را باز می‌گردانیم. از آنجا که الگوریتم دو حلقه‌ی تودرتو دارد که هر کدام دقیقاً به تعداد طول آرایه تکرار می‌شوند، کارایی زمانی آن دقیقا می‌باشد و کارایی فضایی آن زیرا متغیر بزرگی اضافه بر ورودی ایجاد نمی‌کند.

**Algorithm: MajorityElementBruteForce(*A*[0…*n* – 1])**

// compares all elements two by two and counts duplicates to find majority element

// Input: array *A* with *n* elements

// Output: returns majority element if one exists, else returns *None*

*majority* ← *None*

**for** *i* ← 0 **to** *n* – 1 **do**

*count* ← 0

**for** *j* ← 0 **to** *n* – 1 **do**

**if** *A*[*i*] = *A*[*j*] **then** *count* ← *count* + 1

**if** *count* > int(*n* / 2) **then** *majority* ← *A*[*i*]

**return** *majority*

**9 – ب)** در بخش الف با الگوریتمی ساده‌اندیشانه این سوال حل شد، اما در آن الگوریتم بسیاری از مقایسات چند بار تکرار شده و کارایی زمانی بی علت افزایش میافت. راهکاری هوشمندانه‌تر این است که به خاصیت عنصر اکثریت دقت کنیم. همانطور که گفته شد، در یک آرایه تنها یک عنصر اکثریت می‌تواند وجود داشته باشد که تعداد تکرار آن حتماً از بقیه‌ی عناصر بیشتر است. پس اگر هر تکرار یک عنصر را با هر نابرابری یک عنصر دیگر با آن خط زده و کنار بگذاریم، عنصر اکثریت حتماً تا آخر باقی می‌ماند.

این همان الگوریتم مور و بویر است که با پیاده‌سازی آن در قدم اول و بررسی عنصر اکثریت بودن نتیجه (چون ممکن است در آرایه عنصر اکثریت ناموجود باشد) در قدم دوم به کمک مقایسه‌ی تعداد تکرارهایش با نصف پارامتر، پاسخ را میابیم. کارایی زمانی الگوریتم دو بخشی حاصل خطی‌ست زیرا هر بخشِ مستقل با حلقه‌ای به تعداد طول آرایه تکرار می‌شود پس .

**Algorithm: MajorityElementLinear(*A*[0…*n* – 1])**

// finds majority element in linear time using Boyer and Moore’s algorithm

// Input: array *A* with *n* elements

// Output: returns majority element if one exists, else returns *None*

*majority* ← *None* // the candidate for having most count

*repeat* ← 0

**for** *i* ← 0 **to** *n* – 1 **do**

**if** *A*[*i*] = *majority* **then** *repeat* ← *repeat* + 1

**else** *repeat* ← *repeat* – 1

**if** *repeat* = 0 **then** // candidate was cancelled out so pick new candidate

*majority* ← *A*[*i*]

*repeat* ← 1

*count* ← 1 // now we count the repetitions of majority candidate

**for** *i* ← 0 **to** *n* – 1 **do**

**if** *A*[*i*] = *majority* **then** *count* ← *count* + 1

**if** *count* > int(*n* / 2) **then return** *majority*

**else return** *None*

**9 – پ)** با تعمیم الگوریتم استفاده شده در بخش ب می‌توان این مسئله را حل کرد. می‌دانیم حداکثر سه عنصر در آرایه وجود دارند که تعداد تکرارشان بزرگتر از یک چهارم است. پس با تشکیل لیستی سه عضوی همزمان روی سه کاندید عملیات انجام می‌دهیم. به مانند قبل در آرایه پیش رفته و اگر به عنصری مانند یکی از کاندیدها برخوردیم، به شمارنده‌ی آن اضافه می‌کنیم؛ تفاوت در اینجاست که اگر عنصر جدید بود یا آن را به عنوان کاندید جدید قرار می‌دهیم یا (در صورت پر بودن لیست سه عضوی) از شمارنده‌ی هر سه کاندید یکی کم می‌کنیم و اگر شمارنده‌ی کاندیدی صفر شد، آن را از لیست کاندیدها حذف می‌کنیم.

در آخر سه کاندیدی که در لیست باقی خواهند ماند پر تکرارترین عناصر آرایه هستند که با پیش رفتن دوباره در آرایه‌ی اصلی، تعداد تکرار هر کدام را محاسبه کرده و درصورت برقراری شرط، آن‌ها را برمی‌گزینیم. از آنجا که این الگوریتم در نهایت، دو حلقه‌ی تودرتو داشته که یکی به اندازه‌ی طول آرایه و دیگری به تعداد 3 کاندید تکرار می‌شود، کارایی زمانی برابر است با . همچنین متغیر بزرگی اضافه بر ورودی تشکیل نشده است پس کارایی فضایی برای هر مقدار پارامتر همواره ثابت است. شبه کد در صفحه‌ی بعد قابل مشاهده است.

**Algorithm: K-MajorityElementLinear(*A*[0…*n* – 1])**

// finds elements with counts more than *n* / 4 using Boyer and Moore’s algorithm

// Input: array *A* with *n* elements

// Output: returns list of at most three elements, or empty if none found

*candidates* ← [*None*, *None*, *None*]

*repeats* ← [0, 0, 0]

**for** *i* ← 0 **to** *n* – 1 **do**

*new* ← *True*

**for** *j* ← 0 **to** 2 **do** // if element is a repeat of candidates

**if** *A*[*i*] = *candidates*[*j*] **then**

*repeats*[*j*] ← *repeats*[*j*] + 1

*new* ← *False*

**if** *new* **then** // if element is not a repeat of candidates

*filled* ← *True*

**for** *j* ← 0 **to** 2 **do** // if it can be made new candidate

**if** *candidates*[*j*] **is** *None* **then**

*candidates*[*j*] ← *A*[*i*]

*repeats*[*j*] ← 1

*filled* ← *False*

**break** // new candidate assigned so stop iteration

**if** *filled* **then** // if we already have three candidates

**for** *j* ← 0 **to** 2 **do**

*repeats*[*j*] ← *repeats*[*j*] – 1

**if** *repeats*[*j*] = 0 **then** // if candidate is cancelled out

*candidates*[*j*] ← *None*

*results* ← list() // now we count the repetitions of candidates

**for** *j* ← 0 **to** 2 **do**

*count* ← 0

**for** *i* ← 0 **to** *n* – 1 **do**

**if** *A*[*i*] = *candidates*[*j*] **then** *count* ← *count* + 1

**if** *count* > int(*n* / 4) **then**

*results*.append(*candidates*[*j*])

**return** *results*

**10.** یکی از دوستان ما تصمیم گرفته است که سفری کوتاه مدت و کم­هزینه به قاره آفریقا داشته باشد تا از نزدیک به سیر در پهناورترین پارک­های ملی آفریقا بپردازد. او نقشه کامل جاده­های پارک­های ملی آفریقا را تهیه کرده است و می­خواهد تمام جاده­های هر پارک­ ملی را به طور کامل طی کند و در عین حال، زمان و هزینه گردش خود در پارک­های ملی را به حداقل برساند؛ اما با قیدهایی مواجه است:

* از آنجا که حیات وحش آفریقا پر است از انواع درندگان و گوشتخواران و او نمی­خواهد خطر کند، در تمام طول گردش سوار بر خودرو خواهد بود.
* از آنجا که نگران است مردم بومی یا مسافران دیگر، ناقل یک بیماری مسری باشد، او می­خواهد که تنها با راننده (که راهنمای او نیز به حساب می­آید) به گشت در پارک­های ملی بپردازد. پس باید به تنهایی همه هزینه­ یک بار گردش کامل در پارک را بپردازد.

از آنجا که پولی که دوست مسافر ما برای گردش کامل در پارک­ها باید بپردازد نسبتاً زیاد است، به دنبال راهی است که هزینه گشت کامل در هر پارک را به حداقل برساند؛ هزینه گردش او وقتی به حداقل خواهد رسید که مسافت کوتاه­تری را با خودرو طی کند.

آیا می­توانید الگوریتمی قابل­فهم و کارا طراحی کنید تا این دوست را در پیدا کردن کم­هزینه­ترین مسیرهای حرکت در پارک­های ملی کمک کنید؟ اگر فرضاً تعداد جاده­های یک پارک ملی باشد، کارایی الگوریتم‌تان برحسب چقدر خواهد بود؟

**جواب:**

اگر نقشه‌ی پارک را یک گراف بی‌جهت در نظر بگیریم که جاده‌ها همان یال‌ها باشند، فرد نیاز دارد که کمترین مسیر یعنی کمترین تعداد یال را طی کند. برای پیدا کردن مسیر می‌توانیم با اندکی تغییر از روش یافتن دور اویلری در گراف استفاده کنیم.

می‌دانیم اگر گرافی دارای رأس با درجه‌ی فرد باشد، فاقد مسیر اویلری است؛ در نتیجه می‌توانیم با اضافه کردن یال‌های مناسب تکراری به گراف حاصل، چندبار طی کردن جاده را شبیه‌سازی کنیم تا در مسیر اجرای الگوریتم اویلری به مشکل برنخوریم. بعد از آماده شدن گراف با استفاده از الگوریتم Fleury مسیر مورد نظر جهت دور اویلری را به دست می‌آوریم. مراحل کلی اجرای الگوریتم به شکل زیر است:

1. برای هر رأس ابتدا فرد یا زوج بودن درجه رأس (تعداد یال‌های متصل به آن) را برسی می‌کنیم تا از وجود یا عدم وجود دور اویلری آگاه شویم. در صورت زوج بودن همه رأس‌ها به مرحله‌ی 3 می‌رویم و در غیر این صورت (یعنی اگر رأسی درجه‌اش فرد بود)، مرحله‌ی 2 را اجرا می‌کنیم و سپس مرحله‌ی 3 را.
2. با استفاده از ترفند زیر، یال‌هایی به گراف اضافه می‌کنیم تا دارای مسیر اویلری شود:
   1. برای هر رأس در گراف، اگر از درجه فرد بود و یک همسایه با درجه فرد داشت، یال بین آن دو را به گراف اضافه و تکرار می‌کنیم؛ اگر همسایه فرد نداشت به مرحله بعد می‌رویم.
   2. رأس با درجه فرد را با یک یال به رأسی با درجه‌ی زوج که به آن متصل است باز وصل می‌کنیم.
3. با شروع از رأسی به برسی یال‌های مناسب بعدی می‌پردازیم، دو حالت برای مناسب بودن وجود دارد:
   1. یال تنها یالِ متصل به رأس باشد (در مراحل بعدی ممکن است این حالت پیش بیاید)؛
   2. حذف شدن یال بر در دسترس بودن رأس‌های دیگر تأثیر منفی می‌گذارد.
4. ابتدا یال مناسب یافت شده را خروجی می‌دهیم، سپس رأس مبدأ را علامت گذاری می‌کنیم و آن را به رأس انتهایی یال انتخاب شده تغییر می‌دهیم و دوباره به مرحله‌ی 3 بر می‌گردیم.

لازم به ذکر است که منظور از ایجاد یال جدید، احداث جاده نبوده و انتخاب جاده‌ای موجود برای باز گذر کردن از آن است. کد پایتون الگوریتم در پیوست 3 قابل مشاهده است و توضیحات مخصوص هر بخش از الگوریتم در کامنت‌های کدِ پیوست شده آورده شده است. از آنجایی که بخش EulerCircuitHelp نوعی از تابع DFS می‌باشد و درون آن نیز از NextEdgeValid که خود شکلی از DFS بوده استفاده شده است، کارایی زمانی الگوریتم برای رأس‌ها و تقاطع‌های به تعداد*n* و یال‌ها و جاده‌های *m* برابر با O((*n*+*m*)2) می‌باشد.

حالتی کامل‌تر نیز برای حل این سوال وجود دارد. یعنی اگر برای طی کردن هر جاده از مسیر هزینه‌ی متفاوتی داشته باشیم، می‌توانیم ابتدا نقشه پارک را به شکل یک گراف بی‌جهتِ وزن‌دار شبیه‌سازی کنیم (که وزن هر یال برابر با طول آن جاده باشد) و سپس از الگوریتم زیر استفاده کنیم:

1. بررسی می‌کنیم که آیا گراف شامل دور اویلری‌ست یا نه (یعنی درجه‌ی همه‌ی رئوس آن زوج است یا خیر). اگر پاسخ بله است، به مرحله‌ی 3 پرش می‌کنیم. در غیر این صورت به مرحله‌ی زیر می‌رویم و پس از تکمیل آن مرحله‌ی 3 را اجرا می‌کنیم.
2. با ترفند زیر، درجه‌ی تمام رئوس گراف را زوج می‌کنیم:
   1. رئوس فرد را لیست کرده و برای تمام جفت رأس‌های ممکن از رئوس فرد لیست ایجاد می‌کنیم.
   2. برای هر جفت از رئوس، با در نظر گرفتن تمام حالات ممکن (یعنی بررسی وزن یال‌های موجود متصل به آن و رئوسی که هر کدام به آن متصل هستند) کم هزینه‌ترین یال‌ها و راه برای متصل کردن آنها را به دست می‌آوریم.
   3. مجموعه جفت‌های رأس منحصر به فرد با کمترین هزینه‌ی ممکن را پیدا کرده و با اضافه کردن یال‌های موجود کم هزینه‌ی یافت شده، گراف را کامل می‌کنیم تا دارای دور اویلری باشد.
3. با استفاده از الگوریتم Fleury (مراحل 3 و 4 ذکر شده در الگوریتم قبل) مسیر اویلری را برای گراف به دست می‌آوریم و همراه با جمع وزن یال‌ها (هزینه سفر) به عنوان خروجی برمی‌گردانیم.

تفاوت این الگوریتم با الگوریتم قبل در مرحله‌ی دوی آن‌هاست که در گراف وزن‌دار نیاز است وزن یال (جاده‌ای) که برای تکرار انتخاب می‌کنیم کمترین مقدار ممکن باشد. به علت شباهت (و البته طولانی بودن و پیچیدگی بیش از حد مرحله‌ی دوم)، کد این الگوریتم دیگر نوشته نشده است. کارایی زمانی آن نیز وابسته به همان مرحله‌ی دوم، بیشتر از الگوریتم مختص به گراف بی‌وزن بوده و سه حلقه‌ی تودرتوی مورد نیاز برای حل مرحله‌ی دو، کارایی را از مرتبه‌ی مکعبی می‌کند.

**پیوست 1**

# ANALYSIS OF BACKPACK ALGORITHM (question 1)

def Backpack(capacity, n, weights, values):

matrix = [[0 for w in range(capacity + 1)] for i in range(n + 1)]

SelectedItems = []

for i in range(n + 1):

for w in range(capacity + 1):

if i == 0 or w == 0:

matrix[i][w] = 0

elif weights[i – 1] <= w:

previous = matrix[i - 1][w]

new = values[i – 1] + matrix[i – 1][w – weights[i – 1]]

matrix[i][w] = max(previous, new)

else:

matrix[i][w] = matrix[i – 1][w]

totval = matrix[n][capacity]

w = capacity

for i in range(n, 0, – 1):

if totval <= 0:

break

if totval == matrix[i – 1][w]:

continue

else:

SelectedItems.append(weights[i – 1])

totval = totval – values[i – 1]

w = w – weights[i – 1]

return SelectedItems

lengths = list()

for i in range(540, 561):

lengths.append(i)

import random

import timeit

import csv

csvlines = [['Length (n)', 'Time (second)']]

code = '''

from \_\_main\_\_ import Backpack

from \_\_main\_\_ import capacity

from \_\_main\_\_ import n

from \_\_main\_\_ import weights

from \_\_main\_\_ import values'''

for l in lengths:

weights = random.sample(range(l), l)

values = random.sample(range(l), l)

capacity = int(sum(weights)/2)

n = l

time = timeit.timeit(setup = code,

stmt = 'Backpack(capacity, n, weights, values)',

number = 1)

time = round(time, 3)

csvlines.append([l, time])

with open('Q1.csv', 'w', newline='') as csvfile:

writer = csv.writer(csvfile)

writer.writerows(csvlines)

# Results extracted from program run on:

# Python 3.9.2; Windows 10; AMD FX-9830p CPU

# (quad core with single tread and 3.4 G Hertz processing speed).

**پیوست 2**

# ANALYSIS OF RECURSIVE INSERTION SORT (question 2)

def RecursiveInsertionSort(NList, n):

if n <= 1:

return

RecursiveInsertionSort(NList, n – 1)

temp = NList[n – 1]

pick = n – 2

while pick >= 0 and NList[pick] > temp:

NList[pick + 1] = NList[pick]

pick –= 1

NList[pick + 1] = temp

lengths = list()

for i in range(10, 101, 5):

lengths.append(i \* 10)

import timeit

import random

import csv

csvlines = [['Length', 'Time (ms)']]

code = '''

from \_\_main\_\_ import RecursiveInsertionSort

from \_\_main\_\_ import array'''

for l in lengths:

array = random.sample(range(l), l)

time = timeit.timeit(setup = code,

stmt = 'RecursiveInsertionSort(array,len(array))',

number = 1)

time = round(time, 6) \* 1000

csvlines.append([l, time])

with open('recursivetime.csv', 'w', newline='') as csvfile:

writer = csv.writer(csvfile)

writer.writerows(csvlines)

# ANALYSIS OF ITERATIVE INSERTION SORT (question 2)

def InsertionSort(NList):

for pack in range(1, len(NList)):

temp = NList[pack]

pick = pack – 1

while pick >= 0 and NList[pick] > temp:

NList[pick + 1] = NList[pick]

pick –= 1

NList[pick + 1] = temp

lengths = list()

for i in range(10, 101, 5):

lengths.append(i \* 10)

import timeit

import random

import csv

csvlines = [['Length', 'Time (ms)']]

code = '''

from \_\_main\_\_ import InsertionSort

from \_\_main\_\_ import array'''

for l in lengths:

array = random.sample(range(l), l)

time = timeit.timeit(setup = code,

stmt = 'InsertionSort(array)',

number = 1)

time = round(time, 6) \* 1000

csvlines.append([l, time])

with open('nonrecursivetime.csv', 'w', newline='') as csvfile:

writer = csv.writer(csvfile)

writer.writerows(csvlines)

# Results extracted from program run on:

# Python 3.9.2; Windows 10; AMD FX-9830p CPU

# (quad core with single tread and 3.4 G Hertz processing speed).

**پیوست 3**

# Python program to find Eulerian Trail in a given graph (question 10)

from collections import defaultdict

# class to represent an undirected graph

# using adjacency list representation

class Graph:

def \_\_init\_\_(self, vertices):

self.Vertices = vertices # vertices

self.graph = defaultdict(list) # default-dictionary to store graph

self.Time = 0

# method to add edge (u, v) to graph

def addEdge(self, u, v):

self.graph[u].append(v)

self.graph[v].append(u)

# method to remove edge (u, v) from graph

def removEdge(self, u, v):

for index, key in enumerate(self.graph[u]):

if key == v:

self.graph[u].pop(index)

for index, key in enumerate(self.graph[v]):

if key == u:

self.graph[v].pop(index)

# DFS-based method to count reachable vertices from v

def DFSCount(self, v, visited):

count = 1

visited[v] = True

for i in self.graph[v]:

if visited[i] == False:

count = count + self.DFSCount(i, visited)

return count

# method to check if edge (u, v) can be

# considered as next edge in Euler Circuit

def NexEdgeValid(self, u, v):

# the edge (u, v) is valid in one of the following cases:

# 1) if v is the only adjacent vertex of u

if len(self.graph[u]) == 1:

return True

# 2) if there are multiple adjacents, then (u, v) is not a bridge

# do following steps to check if (u, v) is a bridge

else:

# 2 – i) count of vertices reachable from u

visited =[False]\*(self.Vertices)

count1 = self.DFSCount(u, visited)

# 2 – ii) remove edge (u, v) then

# count vertices reachable from u

self.removEdge(u, v)

visited =[False]\*(self.Vertices)

count2 = self.DFSCount(u, visited)

# 2 – iii) add the edge back to the graph

self.addEdge(u, v)

# 2 – iv) if count1 is greater, then edge (u, v) is a bridge

return False if count1 > count2 else True

# method to yield Euler Circuit starting from vertex u

def EulerCircuitHelp(self, u):

# recur for all the vertices adjacent to this vertex

for v in self.graph[u]:

if self.NexEdgeValid(u, v): # if edge (u, v) is not removed

yield ([u, v]) # and is a valid next edge

self.removEdge(u, v)

self.EulerCircuitHelp(v)

# main method to print Eulerian Trail

# it first finds an odd degree vertex (if there is any)

# and then calls EulerCircuitHelp() to return the path

def EulerCircuit(self):

u = 0

for i in range(self.Vertices):

if len(self.graph[i]) % 2 != 0:

u = i

break

return self.EulerCircuitHelp(u)

# main function to find path

def NationalParkTour(Map):

# we must make the map Eulerian by simultaneously

# adding needed duplicate edges to make all vertices even

# these duplicates are the streets that will be travelled twice

for vertex in Map.Vertices:

if len(Map.graph[vertex]) % 2 != 0:

for secondvertex in Map.graph[vertex]:

if len(Map.graph[secondvertex]) % 2 != 0:

Map.addEdge(vertex,secondvertex)

break

else:

continue

if len(Map.graph[vertex]) % 2 != 0:

for secondvertex in Map.graph[vertex]:

Map.addEdge(vertex,secondvertex)

break

else:

continue

Path = list(Map.EulerCircuit())

return Path