فهم الخوارزميات باستفاضة

Grokking Algorithms

دليل مصوَّر للمبرمجين
وغيرهم من الأشخاص الفضوليين

المؤلف

Aditya Y. Bhargava

ترجمة

نخبة من المتخصصين

تواصل معنا

للحصول على المعلومات وطلب هذا الكتاب وغيره من كتب شركة **Coding عربي**

يرجي التواصل على البريد الإلكتروني التالي:

coding.arabic@gmail.com

أو زيارة الصفحة على Facebook من خلال الرابط التالي:

www.facebook.com/coding.arabic

معلومات حول الشركة

تهدف شركتنا إلى إثراء المحتوي العلمي باللغة العربية في مجال الحوسبة الرقمية Computing.

من خلال العمل على الترجمة الدقيقة بواسطة نخبة من المتخصصين في هذا المجال بالتعاون مع الناشرين الأصليين.

تنبيه هام

جميع الحقوق محفوظة لشركة **Coding عربي** للترجمة والنشر.

لا يجوز إعادة إنتاج أي جزء من هذا الكتاب أو تخزينه في نظام استرجاع أو نقله بأي شكل أو بوسائل إلكترونية أو ميكانيكية أو بالنسخ الضوئي أو غير ذلك، دون إذن كتابي مسبق من الشركة.

معلومات عن الترجمة

- تهدف الترجمة للعربية إلى إتاحة المواد العلمية للجمهور العربي بلغتهم الأم وذلك لتعزيز الفهم والإدراك والوعي لدي الناطقين بالعربية بعدما أثبتت الأبحاث جدوى التعلم باللغة الأم مع مراعاة مواكبة العصر عن طريق ذكر المصطلحات الفنية باللغة الإنجليزية.
- الحرص على تقديم المصطلحات الفنية الأساسية باللغة الإنجليزية بجانب الترجمة العربية وذلك بهدف تعزيز قدرة القارئ على التعلم والبحث والقراءة في المواد باللغة الإنجليزية بجانب التوضيح للمفاهيم باللغة العربية.
- تم تقديم بعض الكلمات غير الفنية باللغة الإنجليزية بغرض توضيح وتأكيد المعني وعدم الالتباس عند القارئ في بعض السياقات الخاصة.
- مراعاة تكرار المصطلحات الإنجليزية بشكل مناسب يهدف إلى تثبيت هذه المصطلحات في عقل القارئ.
- مراعاة دمج المصطلحات الإنجليزية خلال النص العربي مما يسهل عملية القراءة والإدراك بدون عناء الرجوع إلى فهرس خاص أو فصل المحتوي العربي عن المصطلحات الفنية الإنجليزية.
- تم في بعض الأحيان توفير ترجمات متعددة لبعض المصطلحات بهدف توضيح أكثر للمعني وإثراء الإدراك والتأكد من وصول المعاني الصحيحة لذهن القارئ.
- تمت دراسة بدائل الترجمة بدقة شديدة ومراجعة للمصادر المتاحة.
- تم بعض المعالجة للنص ليصبح ملائما للقارئ العربي مع حذف بعض العبارات غير الواضحة أو إعادة صياغتها للمساعدة في دعم الفهم الجيد لدي القارئ.

التقييم والأخطاء

نتمنى دائما التواصل المستمر مع عملاءنا للتعبير عن آرائهم في منتجاتنا المنشورة.

وإذا واجهت أي خطأ مطبعي أو ترجمة معينة أو خطأ في الأكواد الملحقة بهذا الكتاب.

يرجي التواصل على البريد الإلكتروني التالي:

coding.arabic@gmail.com

أو التواصل من خلال الصفحة على Facebook من خلال الرابط التالي:

www.facebook.com/coding.arabic

المحتويات

مقدمة 9

حول هذا الكتاب 10

خارطة الطريق Roadmap 10

كيفية استخدام هذا الكتاب 12

من ينبغي أن يقرأ هذا الكتاب 12

1 مقدمة إلى الخوارزميات Algorithms 13

مقدمة 13

ماذا سوف تتعلم عن الأداء Performance 14

ماذا سوف تتعلم عن حل المسائل Problem Solving 14

البحث الثنائي Binary Search 15

طريقة أفضل للبحث 16

وقت التشغيل Running Time 21

التدوين Big O Notation 22

أوقات تشغيل الخوارزمية Algorithm Running Times 22

تنمو بمعدلات مختلفة

تصوّر Visualizing أوقات تشغيل Big O المختلفة 25

يُنشئ Big O وقت تشغيل أسوأ حالة Worst-Case Running Time 27

بعض أوقات تشغيل Big O الشائعة 27

مندوب المبيعات المتنقل Travelling Salesperson 29

الخلاصة 31

2 ترتيب التحديد Selection Sort 32

كيف تعمل الذاكرة Memory 33

المصفوفات Arrays والقوائم المرتبطة Linked Lists 34

القوائم المرتبطة Linked Lists 36

المصفوفات Arrays 37

المصطلحات Terminology 38

الإدراج Inserting في منتصف القائمة List 39

عمليات الحذف Deletions 41

ترتيب التحديد Selection Sort 43

الخلاصة 48

3 التكرارية Recursion 49

التكرارية Recursion 50

الحالة الأساسية Base Case والحالة التكرارية Recursive Case 52

الدفتر Stack 54

دفتر الاستدعاءات Call Stack 55

دفتر الاستدعاءات Call Stack مع التكرارية Recursion 57

الخلاصة 61

4 الترتيب السريع Quicksort 62

فرِّق تَسُد Divide & Conquer 63

الترتيب السريع Quicksort 70

إعادة النظر في تدوين Big O Notation 76

ترتيب الدمج Merge Sort في مقابل الترتيب السريع Quicksort 77

الحالة المتوسطة Average Case في مقابل أسوأ حالة Worst Case 79

الخلاصة 83

5 جداول التجزئة Hash Tables 84

دوال التجزئة Hash Functions 87

حالات الاستخدام Use Cases 91

استخدام جداول التجزئة لعمليات البحث Lookups 91

منع الإدخالات المتكررة Duplicate Entries 92

استخدام جداول التجزئة Hash Tables كذاكرة تخزين مؤقت Cache 94

الخلاصة 97

التصادمات Collisions 97

الأداء Performance 100

عامل التحميل Load Factor 102

دالة تجزئة Hash Function جيدة 104

الخلاصة 106

6 بحث الاتساع-أولاً Breadth-First Search 107

مقدمة إلى الرسوم البيانية Graphs 108

ما هو الرسم البياني Graph؟ 110

بحث الاتساع أولاً Breadth-First Search 111

إيجاد أقصر مسار Shortest Path 113

قوائم الانتظار Queues 115

تنفيذ الرسم البياني Implementing Graph 117

تنفيذ الخوارزمية Implementing Algorithm 119

وقت التشغيل Running Time 124

الخلاصة 127

Coding عربي

7 خوارزمية ديكسترا Dijkstra's Algorithm 128

العمل مع خوارزمية ديكسترا Dijkstra Algorithm 129
المصطلحات Terminology 134
المقايضة للحصول على البيانو Trading For Piano 137
حواف ذات وزن - ترجيح سالب Negative-Weight Edges 143
التنفيذ Implementation 146
الخلاصة 157

8 الخوارزميات الطامعة - الشَّرِهَة Greedy Algorithms 158

مسألة جدولة قاعات الدراسة Classroom Scheduling Problem 158
مسألة حقيبة الظهر Knapsack Problem 161
مسألة تغطية المجموعات The Set-Covering Problem 163
خوارزميات التقريب Approximation Algorithms 165
مسائل NP-Complete 170
مندوب المبيعات المتنقل Traveling Salesperson، خطوة بخطوة 171
كيف يمكنك معرفة ما إذا كانت المسألة NP-Complete؟ 175
الخلاصة 178

9 البرمجة الديناميكية Dynamic Programming 179

مسألة حقيبة الظهر The Knapsack Problem 179
الحل البسيط The Simple Solution 180
البرمجة الديناميكية Dynamic Programming 181
الأسئلة الشائعة حول مسألة حقيبة الظهر Knapsack Problem FAQ 189
ماذا يحدث إذا قمت بإضافة عنصر Add Item؟ 189
ماذا يحدث إذا قمت بتغيير ترتيب الصفوف Order of Rows؟ 192
هل يمكنك ملء الشبكة Fill Grid من حيث الأعمدة Column-Wise 193

بدلاً من الصفوف Row-Wise؟
ماذا يحدث إذا قمت بإضافة عنصر أصغر Add Smaller Item؟ 193
هل يمكنك سرقة كسور Fractions من عنصر Item؟ 193
تحسين Optimizing خط سير سفرك 194
التعامل مع العناصر Items التي تعتمد على بعضها البعض 195
هل من الممكن أن يتطلب الحل أكثر من حقيبتي ظهر فرعيتين Sub-Knapsacks؟ 196
هل من الممكن ألا يملأ الحل الأفضل Best Solution حقيبة الظهر بالكامل؟ 196
أطول سلسلة حروف فرعية مشتركة Longest Common Substring 197
عمل الشبكة Making Grid 198
ملء الشبكة Filling Grid 199
الحل Solution 200
أطول متتالية مشتركة Longest Common Subsequence 201
أطول متتالية مشتركة - الحل 202
الخلاصة 205

10 أقرب عدد K من الجيران K-Nearest Neighbors 206

تصنيف البرتقال Oranges في مقابل الجريب فروت Grapefruit 206
بناء نظام اقتراحات Recommendations System 208
استخراج الملامح Feature Extraction 210
الانحدار Regression 214
انتقاء الملامح الجيدة Picking Good Features 216
مقدمة في التعلم الآلي Introduction To Machine Learning 217
التعرف البصري على الحروف OCR 218
بناء فلتر البريد العشوائي Spam Filter 219
توقع سوق الأسهم Predicting Stock Market 220
الخلاصة 220

الأشجار Trees 221
الفهارس المعكوسة Inverted Indexes 225
تحويل فورييه The Fourier Transform 225
الخوارزميات المتوازية Parallel Algorithms 226
MapReduce 227
لماذا الخوارزميات الموزَّعة Distributed Algorithms مفيدة؟ 227
الدالة Map 228
الدالة Reduce 229
فلاتر - مُرشِّحات بلوم Bloom Filters و HyperLogLog 229
فلاتر بلوم Bloom Filters 231
خوارزميات HyperLogLog 231
خوارزميات SHA 232
مقارنة الملفات Comparing Files 232
التحقق من كلمات المرور Checking Passwords 234
التجزئة الحساسة للموضع Locality-Sensitive Hashing 235
تبادل مفاتيح ديفي-هيلمان Diffie-Hellman Key Exchange 236
البرمجة الخطية Linear Programming 237
الخاتمة 238
الإجابات على التمارين Answers To Exercises 239

Coding عربي

مقدمة

دخلت البرمجة Programming لأول مرة كهواية. علَّمني كتاب Visual Basic 6 For Dummies الأساسيات، وواصلت قراءة الكتب لمعرفة المزيد. لكن موضوع الخوارزميات Algorithms كان لا يمكن اختراقه بالنسبة لي. أتذكر تصفح جدول المحتويات لكتابي الأول عن الخوارزميات، معتقدًا "سأفهم هذه الموضوعات أخيرًا!" لكنها كانت أشياء كثيفة، واستسلمت بعد بضعة أسابيع. عندما حصلت على أول أستاذ جيد للخوارزميات أدركت كم كانت هذه الأفكار بسيطة وأنيقة.

قبل بضع سنوات، كتبت أول منشور مدونة مصور Illustrated Blog Post. أنا متعلم بصري Visual Learner، وقد أحببت حقًا الأسلوب المصور Illustrated Style. منذ ذلك الحين، قمت بكتابة بعض المنشورات المصورة حول البرمجة الدالِّية Functional Programming، وGit، والتعلم الآلي Machine Learning، والتزامن Concurrency. بالمناسبة: كنت كاتبًا متواضعًا عندما بدأت مسيرتي. من الصعب شرح المفاهيم الفنية – التقنية Technical Concepts. يستغرق الخروج بأمثلة جيدة وقتًا، وشرح مفهوم صعب يستغرق وقتًا. لذلك فمن الأسهل التغاضي عن الأشياء الصعبة. اعتقدت أنني أقوم بعمل جيد، إلى أن أصبحت إحدى منشوراتي مشهورة، جاء إلى زميل في العمل وقال، "لقد قرأت منشورك وما زلت لا أفهم هذا." لا يزال لدي الكثير لأتعلمه عن الكتابة.

في مكان ما أثناء كتابة منشورات المدونة Blog Posts هذه، اتصلت بي مؤسسة مانينغ Manning للنشر وسألت عما إذا كنت أرغب في كتابة كتاب مصور Illustrated Book. حسنًا، اتضح أن محرري مانينغ Manning Editors يعرفون الكثير عن شرح المفاهيم التقنية Technical Concepts، وقد علموني كيفية التدريس. لقد كتبت هذا الكتاب لتجربة معينة: أردت أن أكتب كتابًا يشرح الموضوعات التقنية الصعبة جيدًا، وأردت كتابًا سهل القراءة عن الخوارزميات Algorithms. لقد قطعت كتابتي شوطًا طويلاً منذ ذلك المنشور الأول في المدونة، وآمل أن تجد هذا الكتاب سهل القراءة وغني بالمعلومات.

coding عربي

حول هذا الكتاب

تم تصميم هذا الكتاب ليكون سهل المتابعة. أتجنب القفزات الكبيرة في التفكير. في أي وقت يتم تقديم مفهوم جديد، أشرح ذلك على الفور أو أخبرك متى سأشرح ذلك. يتم تعزيز المفاهيم الأساسية من خلال التمارين Exercises والشروحات المتعددة بحيث يمكنك التحقق من افتراضاتك والتأكد من متابعتك لها.

أنا أشرح بالأمثلة. هدفي هو تسهيل تصور هذه المفاهيم. أعتقد أيضًا أننا نتعلم بشكل أفضل من خلال قدرتنا على تذكر شيء نعرفه بالفعل، والأمثلة تجعل الاسترجاع أسهل. لذلك عندما تحاول تذكر الفرق بين المصفوفات Arrays والقوائم المرتبطة Linked Lists (تم توضحيها في الفصل الثاني)، يمكنك فقط التفكير في الجلوس لمشاهدة فيلم. أيضًا، كما هو واضح، فأنا متعلم بصري. إذن هذا الكتاب مليء بالصور.

محتويات الكتاب مرتبة بعناية. ليست هناك حاجة لكتابة كتاب يغطي كل خوارزمية ترتيب Sorting Algorithm - ولهذا السبب لدينا ويكيبيديا Wikipedia وأكاديمية خان Khan Academy. جميع الخوارزميات التي أدرجتها عملية Practical. لقد وجدتها مفيدة في عملي كمهندس برمجيات Software Engineer، وتوفر أساسًا جيدًا لمواضيع أكثر تعقيدًا. قراءة سعيدة!

خارطة الطريق Roadmap

الفصول الثلاثة الأولى من هذا الكتاب تقوم بإرساء الأُسس Foundations:

- **الفصل الأول:** ستتعلم أول خوارزمية عملية Practical Algorithm لك: البحث الثنائي Binary Search. تتعلم أيضًا تحليل سرعة الخوارزمية باستخدام تدوين Big O Notation. يتم استخدام تدوين Big O في جميع أنحاء الكتاب لتحليل مدى بطء أو سرعة الخوارزمية.
- **الفصل الثاني:** ستتعرف على هيكلين أساسيين للبيانات Two Fundamental Data Structures: المصفوفات Arrays والقوائم المرتبطة Linked Lists. تُستخدم هياكل البيانات Data Structures هذه في جميع أنحاء الكتاب، ويتم استخدامها لإنشاء هياكل بيانات أكثر تقدمًا مثل جداول التجزئة Hash Tables (الفصل الخامس).
- **الفصل الثالث:** سوف تتعلم المزيد عن التكرارية Recursion، وهي طريقة مفيدة تستخدمها العديد من الخوارزميات Algorithms (مثل الترتيب السريع Quicksort، تمت تغطيته في الفصل الرابع).

من خلال خبرتي، يُعد تدوين Big O Notation والتكرارية Recursion موضوعات صعبة للمبتدئين. لذا فقد قضيت وقتًا إضافيًا في هذه الأقسام.

يُقدم باقي الكتاب خوارزميات Algorithms ذات تطبيقات واسعة Broad Applications:

- **طرق حل المسائل Problem-Solving Techniques** - تمت تغطيتها في الفصول 4 و 8 و 9. إذا صادفت مسألة Problem ولم تكن متأكدًا من كيفية حلها بكفاءة، فجرّب طريقة فرِّق تَسُد Divide And Conquer (الفصل الرابع) أو البرمجة الديناميكية Dynamic Programming (الفصل التاسع). أو قد تدرك أنه لا يوجد حل فعال Efficient Solution، وتحصل على إجابة تقريبية Approximate Answer باستخدام خوارزمية شرهة - طامعة Greedy Algorithm بدلاً من ذلك (الفصل الثامن).
- **جداول التجزئة Hash Tables** - تمت تغطيتها في الفصل الخامس. جدول التجزئة Hash Table هو هيكل بيانات Data Structure مفيد للغاية. يحتوي على مجموعات Sets من أزواج المفاتيح والقيم Key And Value Pairs، مثل اسم الشخص وعنوان بريده الإلكتروني Email Address، أو اسم المستخدم Username وكلمة المرور المرتبطة به Associated Password.. عندما أريد حل مسألة ما Solve a Problem، فإن اثنان من خطط الهجوم التي أبدأ بها هما "هل يمكنني استخدام جدول التجزئة Hash Table؟" و "هل يمكنني نمذجتها Model على هيئة رسم بياني Graph؟"
- **خوارزميات الرسم البياني Graph Algorithms** - تمت تغطيتها في الفصلين 6 و 7. الرسوم البيانية Graphs هي طريقة لنمذجة Model شبكة Network: شبكة اجتماعية Social Network، أو شبكة طرق، أو خلايا عصبية Neurons، أو أي مجموعة أخرى من الروابط Connections. بحث الاتساع-أولاً (الفصل السادس) وخوارزمية ديكسترا Dijkstra's Algorithm (الفصل السابع) تُعد طرقًا للعثور على أقصر مسافة بين نقطتين في شبكة Network: يمكنك استخدام هذا الأسلوب لحساب درجات الانفصال بين شخصين أو أقصر طريق إلى وجهة معينة.
- **أقرب عدد K من الجيران K-Nearest Neighbors (KNN)** - تمت تغطيته في الفصل العاشر. هذه خوارزمية بسيطة للتعلم الآلي Simple Machine-Learning Algorithm. يمكنك استخدام KNN لإنشاء نظام توصيات – اقتراحات Recommendations System، أو محرك OCR Engine، أو نظام للتنبؤ بقيمة الأسهم Stock Values - أي شيء يتضمن توقُّع قيمة Predicting Value مثل ("نعتقد أن Adit سيقوم بتقييم هذا الفيلم ب 4 نجوم") أو تصنيف كائن Classifying Object ("هذا الحرف هو Q").
- **الخطوات التالية Next Steps** - يتطرق الفصل الحادي عشر إلى 10 خوارزميات Algorithms من شأنها أن تساعد في القراءة المتعمقة الجيدة بعد هذا الكتاب.

كيفية استخدام هذا الكتاب

تم تصميم ترتيب ومحتويات هذا الكتاب بعناية. إذا كنت مهتمًا بموضوع ما، فلا تتردد في القفز إليه. خلاف ذلك، اقرأ الفصول بالترتيب - فهي تبني بعضها على بعض.

أوصي بشدة بتنفيذ أكواد Codes الأمثلة Examples بنفسك. ما عليك سوى كتابة عينات الأكواد Code Samples أو تنزيلها من الرابط التالي:

https://drive.google.com/drive/folders/1xUbc896YaKp2bY89F_aTKGC2N2G4LUe9?usp=sharing

ثم تنفيذها Execute. سوف تكتسب المزيد من المعرفة والخبرة إذا فعلت ذلك.

أوصي أيضًا بإجراء التمارين في هذا الكتاب. التدريبات قصيرة - عادة ما تكون دقيقة أو دقيقتين، وأحيانًا من 5 إلى 10 دقائق. سيساعدونك في التحقق من تفكيرك، حتى تعرف أنك تخرج عن المسار الصحيح قبل أن تذهب بعيدًا.

من ينبغي أن يقرأ هذا الكتاب

يستهدف هذا الكتاب أي شخص يعرف أساسيات البرمجة Coding ويريد فهم الخوارزميات Algorithms. ربما لديك بالفعل مسألة برمجية Coding Problem وتحاول إيجاد حل خوارزمي - حوسبي Algorithmic Solution لها. أو ربما تريد أن تفهم ما هي الخوارزميات المفيدة لأغراض معينة. فيما يلي قائمة قصيرة وغير كاملة بالأشخاص الذين ربما يجدون هذا الكتاب مفيدًا:

- المبرمجون الهواة Hobbyist Coders
- طلاب دورات البرمجة Coding Boot Camp Students
- خريجو علوم الحاسب Computer Science Grads الذين يبحثون عن إعادة تنشيط Refresher
- خريجو الفيزياء Physics والرياضيات Math والخريجين الآخرين المهتمين بالبرمجة Programming

اصطلاحات الأكواد Code Conventions والتنزيلات Downloads

يتم تقديم كل الأكواد الموجودة في الكتاب بخط مثل هذا `fixed-width font like this` لفصله عن النص العادي. تُصاحب التعليقات التوضيحية للكود Code Annotations بعض قوائم الأكواد Code Listings المذكورة في الكتاب، مما يُبرز المفاهيم المهمة.

أعتقد أنك تتعلم بشكل أفضل عندما تستمتع حقًا بالتعلم - لذا استمتع، وقم بتشغيل Run عينات الأكواد Code Samples!

1 مقدمـــــــــة إلى الخوارزميات Algorithms

في هذا الفصل

- تحصل على أساس Foundation لبقية الكتاب.
- تكتب خوارزمية البحث Search Algorithm الأولى (البحث الثنائي Binary Search).
- تتعلم كيفية التحدث عن وقت تشغيل Running Time خوارزمية Algorithm (Big O Notation).
- تتعرف على أسلوب شائع لتصميم الخوارزميات Designing Algorithms (التكرارية Recursion).

مقدمة

الخوارزمية Algorithm هي مجموعة Set من التعليمات Instructions لإنجاز مهمة Task ما. كل جزء من الكود Code يمكن تسميته بالخوارزمية Algorithm، لكن هذا الكتاب يغطي التفاصيل الأكثر إثارة للاهتمام. اخترت خوارزميات لتضمينها في هذا الكتاب لأنها سريعة، أو تحل مسائل Problems مثيرة للاهتمام، أو كلاهما. فيما يلي بعض النقاط البارزة:

- يتحدث الفصل الأول عن البحث الثنائي Binary Search ويوضح كيف يمكن لخوارزمية تسريع Algorithm الكود Code الخاص بك. في أحد الأمثلة، ينخفض عدد الخطوات Steps المطلوبة من 4 بلايين Billions إلى 32 خطوة فقط!
- جهاز GPS يستخدم خوارزميات الرسم البياني Graph Algorithms (كما ستتعلم في الفصول 6 و 7 و 8) لحساب أقصر طريق Shortest Route للوصول إلي وجهتك.
- يمكنك استخدام البرمجة الديناميكية Dynamic Programming (التي تمت مناقشتها في الفصل 9) لكتابة خوارزمية ذكاء اصطناعي AI والتي تقوم بلعب لعبة الداما Checkers.

Coding عربي

في كل حالة، سأقوم بوصف الخوارزمية Algorithm وأعطيك مثالاً. ثم سأتحدث عن وقت تشغيل Running Time الخوارزمية Algorithm في تدوين Big O Notation. أخيرًا، سوف أستكشف أنواع المسائل Problems الأخرى التي يمكن حلها باستخدام نفس الخوارزمية.

ماذا سوف تتعلم عن الأداء Performance

الأخبار السارة هو أنه بعد مجهود كبير أصبح تنفيذ Implementation كل خوارزمية Algorithm في هذا الكتاب متاحًا بلغتك Language المفضلة، وبذلك لا يتعين عليك كتابة كل خوارزمية Algorithm بنفسك! لكن هذه التطبيقات Implementations عديمة الفائدة إذا لم تفهم المفاضلات Trade-Offs (التنازل عن ميزة من أجل الحصول على أخرى). في هذا الكتاب، ستتعلم مقارنة المفاضلات بين الخوارزميات Algorithms المختلفة: هل يجب استخدام ترتيب الدمج Merge Sort أو الترتيب السريع Quicksort؟ هل يجب عليك استخدام مصفوفة Array أم قائمة List؟ مجرد استخدام هيكل بيانات Data Structure مختلف يمكن أن يحدث فرقًا كبيرًا.

ماذا سوف تتعلم عن حل المسائل Problem Solving

ستتعلم أساليب حل المسائل التي ربما كانت خارج متناولك حتى الآن. فمثلا:

- إذا كنت تحب صنع ألعاب الفيديو Video Games، فيمكنك كتابة نظام ذكاء اصطناعي AI System يقوم باتباع Follow المستخدم User باستخدام خوارزميات الرسم البياني Graph Algorithms.
- سوف تتعلم كيفية عمل نظام اقتراحات - توصيات Recommendations System باستخدام خوارزمية أقرب عدد K من الجيران K-Nearest Neighbors.
- بعض المسائل Problems لا يمكن حلها في وقت مناسب Timely Manner! يوضح لك الجزء من هذا الكتاب الذي يتحدث عن مشكلات NP-Complete كيفية تمييز تلك المسائل Problems والتوصل إلى خوارزمية Algorithm تعطيك إجابة تقريبية Approximate Answer.

بشكل عام، بحلول نهاية هذا الكتاب، ستعرف بعض الخوارزميات الأكثر قابلية للتطبيق على نطاق واسع Widely Applicable Algorithms. يمكنك بعد ذلك استخدام معرفتك الجديدة للتعرف على المزيد من الخوارزميات المحددة Specific Algorithms للذكاء الاصطناعي AI وقواعد البيانات Databases وهكذا. أو يمكنك القيام بتحديات أكبر في العمل.

ما تحتاج إلى معرفته

ستحتاج إلى معرفة أساسيات علم الجبر Basic Algebra قبل البدء في هذا الكتاب. على وجه الخصوص، انظر إلى هذه الدالة: $f(x) = x \times 2$. ما هي نتيجة $f(5)$؟ إذا أجبت 10، فأنت جاهز. بالإضافة إلى ذلك، سيكون من السهل متابعة هذا الفصل (وهذا الكتاب) إذا كنت معتادًا على لغة برمجة Programming Language واحدة.

كل الأمثلة في هذا الكتاب موجودة بلغة بايثون Python. إذا كنت لا تعرف أي لغة برمجة وترغب في تعلم لغة واحدة، فاختر Python - فهي رائعة للمبتدئين. إذا كنت تعرف لغة أخرى، مثل روبي Ruby، فستكون بخير.

البحث الثنائي Binary Search

لنفترض أنك تبحث عن شخص في دفتر الهاتف (يا لها من جملة قديمة!). يبدأ اسمه بحرف K. يمكنك أن تبدأ من البداية وتستمر في تقليب الصفحات حتى تصل إلى حروف K. لكن من المرجح أنك ستبدأ من صفحة في المنتصف، لأنك تعلم أن حروف K ستكون بالقرب من منتصف دفتر الهاتف. أو افترض أنك تبحث عن كلمة في القاموس، وتبدأ بالحرف O. مرة أخرى، ستبدأ بالقرب من المنتصف.

لنفترض الآن أنك قمت بتسجيل الدخول إلى Facebook. عندما تفعل ذلك، يتعين على Facebook التحقق من أن لديك حسابًا Account على الموقع Site. لذلك، يحتاج إلى البحث عن اسم المستخدم Username الخاص بك في قاعدة البيانات Database الخاصة به. افترض أن اسم المستخدم الخاص بك هو Karlmageddon. يمكن أن يبدأ Facebook من A ويبحث عن اسمك - ولكن من المنطقي أكثر أن يبدأ في مكان ما في المنتصف.

هذه مسألة بحث Search Problem. وجميع هذه الحالات Cases تستخدم نفس الخوارزمية Algorithm لحل المسألة Solve Problem: البحث الثنائي Binary Search.

البحث الثنائي Binary Search هو خوارزمية Algorithm. مدخلاتها Input عبارة عن قائمة مرتَّبة Sorted List من العناصر Elements (سأشرح لاحقًا سبب الحاجة إلى الترتيب Sort). إذا كان العنصر Element الذي تبحث عنه موجودًا في تلك القائمة List، سيقوم بإرجاع Return البحث الثنائي Binary Search الموضع Position الذي يقع فيه. وإلا، فإن البحث الثنائي Binary Search يقوم بإرجاع Returns لا شيء Null.

علي سبيل المثال:

البحث عن الشركات في دليل الهاتف بواسطة البحث الثنائي

فيما يلي مثال على كيفية عمل البحث الثنائي Binary Search. أقوم بالتفكير في رقم بين 1 و 100.

عليك محاولة تخمين الرقم في أقل عدد ممكن من المحاولات. مع كل تخمين، سأخبرك ما إذا كان تخمينك منخفضًا جدًا Too Low أم مرتفعًا جدًا Too High أم صحيحًا Correct. لنفترض أنك بدأت في التخمين مثل هذا: 1 ، 2 ، 3 ، 4 .... وإليك كيف ستسير الأمور.

منهج سيء لتخمين الأرقام

هذا بحث بسيط Simple Search (ربما يكون المصطلح الأفضل هو البحث الغبي). مع كل تخمين، فإنك تقوم باستبعاد Eliminate رقم واحد فقط. إذا كان الرقم هو 99، فقد يستغرق الأمر 99 تخمينًا للوصول إلى هناك!

طريقة أفضل للبحث

فيما يلي طريقة أفضل. ابدأ بالرقم 50.

منخفض جدًا Too Low، لكنك قمت باستبعاد نصف الأرقام! أنت تعلم الآن أن الأرقام من 1 إلى 50 كلها منخفضة جدًا. التخمين التالي: 75.

مرتفع جدًا Too High، لكنك مرة أخرى قمت باستبعاد نصف الأرقام المتبقية! باستخدام البحث الثنائي Binary Search، يمكنك تخمين الرقم الأوسط Middle Number واستبعاد نصف الأرقام المتبقية في كل مرة. الرقم التالي هو 63 (في المنتصف بين 50 و 75).

هذا بحث ثنائي Binary Search. لقد تعلمت للتو أول خوارزمية Algorithm! فيما يلي عدد الأرقام التي يمكنك استبعادها في كل مرة.

تخلص من نصف الأرقام في كل مرة باستخدام البحث الثنائي

مهما كان الرقم الذي أفكر فيه، يمكنك التخمين بحد أقصى Maximum سبع تخمينات Guesses - لأنك تستبعد الكثير من الأرقام مع كل تخمين!

لنفترض أنك تبحث عن كلمة في القاموس. يحتوي القاموس على 240.000 كلمة. في أسوأ حالة Worst Case، كم عدد الخطوات التي تعتقد أن كل بحث Search سيتخذها؟

قد يقوم البحث البسيط باتخاذ 240 ألف خطوة إذا كانت الكلمة التي تبحث عنها هي آخر كلمة في القاموس. مع كل خطوة Step من البحث الثنائي Binary Search، تقوم بتقليص عدد الكلمات إلى النصف حتى يتبقى لك كلمة واحدة فقط.

لذا سوف يقوم البحث الثنائي باتخاذ 18 خطوة - فرق كبير! بشكل عام، بالنسبة لأي قائمة List عدد عناصرها n، سوف يقوم البحث الثنائي باتخاذ Binary Search خطوات عددها Log_2 n للتشغيل Run في أسوأ حالة Worst Case، بينما يقوم البحث البسيط Simple Search باتخاذ خطوات عددها n.

اللوغاريتمات Logarithms

قد لا تتذكر ما هي اللوغاريتمات، ولكن ربما تعرف ما هي الأُسيَّات (مشتقة من أُسْ) Exponentials.

Log_{10} 100 مثل السؤال، "كم عدد العشرات 10 التي نضربها Multiply معًا لنحصل على 100؟" الجواب هو 2 :(10x10). إذن Log_{10} 100 = 2. اللوغاريتمات Logs هي مقلوب الأُسيَّات Exponentials.

اللوغاريتمات Logs هي مقلوب الأُسيَّات Exponentials

في هذا الكتاب، عندما أتحدث عن وقت التشغيل Running Time في تدوين Big O Notation (يتم شرحه لاحقًا)، Log تعني دائمًا Log_2. عندما تبحث Search عن عنصر Element باستخدام بحث بسيط Simple Search، في أسوأ حالة Worst Case قد تضطر إلى النظر إلى كل عنصر على حدة. لذلك لقائمة List من 8 أرقام، يجب عليك التحقق Check من 8 أرقام على الأكثر. للبحث الثنائي Binary Search، يجب عليك التحقق

من عناصر عددها log n في أسوأ حالة. لقائمة List من 8 عناصر، 3 == 8 log، لأن 8 == 2^3. لذلك بالنسبة لقائمة List من 8 أرقام، يجب عليك التحقق من 3 أرقام على الأكثر. لقائمة من 1024 عنصرًا Element، 10 = 1024 log لأن 1024 == 2^{10}. لذلك لقائمة من 1024 رقمًا، يجب عليك التحقق من 10 أرقام على الأكثر.

ملاحظة

سأتحدث كثيرًا عن الوقت اللوغاريتمي Log Time في هذا الكتاب، لذا يجب أن تفهم مفهوم اللوغاريتمات Logarithms. إذا لم تكن كذلك، فإن أكاديمية خان Khan Academy (khanacademy.org) لديها مقطع فيديو جميل يوضح ذلك.

ملاحظة

ينجح البحث الثنائي Binary Search فقط عندما تكون قائمتك List مرتبة Sorted. على سبيل المثال، يتم ترتيب Sort الأسماء في دفتر الهاتف بالترتيب الأبجدي، بحيث يمكنك استخدام البحث الثنائي Binary Search للبحث عن اسم. ماذا سيحدث إذا لم يتم ترتيب Sort الأسماء؟

دعونا نرى كيفية كتابة بحث ثنائي Binary Search بلغة بايثون Python. عينة الكود Code Sample هنا تستخدم المصفوفات Arrays. إذا كنت لا تعرف كيفية عمل المصفوفات Arrays، فلا داعي للقلق؛ سيتم تناولها في الفصل التالي. تحتاج فقط إلى معرفة أنه يمكنك تخزين تسلسل Sequence من العناصر Elements في صف Row من الخانات المتتالية تسمى مصفوفة Array. يتم ترقيم الخانات بدءًا من 0: الخانة الأولى في الموضع Position رقم 0، والثانية رقم 1، والثالثة رقم 2، وهكذا.

الدَّالة `binary_search` تأخذ مصفوفة مرتَّبة Sorted Array وعنصرًا Item. إذا كان العنصر موجودًا في المصفوفة Array، فإن الدالة Function تقوم بإرجاع Return موضعها Position. سوف تتعقب Track أي جزء من المصفوفة Array يجب عليك البحث خلاله. في البداية، هذه هي المصفوفة بأكملها:

```
low = 0
high = len(list) - 1
```

في كل مرة تتحقق Check من العنصر الأوسط Middle Element:

```
mid = (low + high) / 2
guess = list[mid]
```

يتم تقريب mid تلقائيًا بواسطة Python إذا لم يكن (low + high) عددًا زوجيًا Even Number

إذا كان التخمين Guess منخفضًا جدًا Too Low، فأنت تقوم بتحديث المتغير Low وفقًا لذلك:

```
if guess < item:
  low = mid + 1
```

وإذا كان التخمين مرتفعًا جدًا Too High، فأنت تقوم بتحديث المتغير high. ها هو الكود Code الكامل:

```
def binary_search(list, item):
  low = 0
  high = len(list)-1

  while low <= high:
    mid = (low + high)
guess = list[mid]
    if guess == item:
      return mid
    if guess > item:
      high = mid - 1
    else:
      low = mid + 1
  return None

my_list = [1, 3, 5, 7, 9]
```

Low و High يقومان بتتبع أي جزء من القائمة سوف تبحث فيه

بينما لم تقم بحصره في عنصر واحد ...

... تحقق من العنصر الأوسط.

إذا تم العثور على العنصر

إذا كان التخمين مرتفعًا جدًا Too High

إذا كان التخمين منخفضا جدًا Too Low

إذا كان العنصر غير موجود

دعونا نختبرها Test!

```
print binary_search(my_list, 3) # => 1
print binary_search(my_list, -1) # => None
```

تذكر أن القوائم Lists تبدأ من 0
تحتوي الخانة Slot الثانية على
رقم تسلسلي (Index) 1.

None تعني لا شيء في بايثون Python.
تشير إلى أنه لم يتم العثور على العنصر Item.

التمارين Exercises

1.1 افترض أن لديك قائمة مرتبة Sorted List مكونة من 128 اسمًا، وأنت تبحث خلالها Searching Through باستخدام البحث الثنائي Binary Search. ما هو الحد الأقصى Maximum لعدد الخطوات Steps التي ستتخذها؟

1.2 افترض أنك ضاعفت Doubled حجم القائمة List Size. ما هو الحد الأقصى Maximum لعدد الخطوات Steps الآن؟

وقت التشغيل Running Time

في أي وقت أتحدث فيه عن خوارزمية Algorithm، سأناقش وقت تشغيلها Running Time. بشكل عام، تريد اختيار الخوارزمية الأكثر فاعلية Efficient - سواء كنت تحاول التحسين Optimize للوقت Time أو المساحة Space.

بالعودة إلى البحث الثنائي Binary Search. كم من الوقت توفره باستخدامه؟ حسنًا، كان النهج Approach الأول هو التحقق Check من كل رقم Number، واحدًا تلو الآخر. إذا كانت هذه قائمة List من 100 رقم، فستحتاج إلى ما يصل إلى 100 تخمين Guess. إذا كانت قائمة List من 4 مليارات رقم، فإنها تتطلب ما يصل إلى 4 مليارات تخمين. لذا فإن الحد الأقصى Maximum لعدد التخمينات هو نفس حجم القائمة. هذا يسمى الوقت الخَطِّي Linear Time.

البحث الثنائي مختلف Binary Search. إذا كانت القائمة تتكون من 100 عنصر، فستستغرق 7 تخمينات على الأكثر. إذا كانت القائمة تحتوي على 4 مليارات عنصر، فستأخذ 32 تخمينًا على الأكثر. هذا البحث الثنائي قوي، أليس كذلك؟ البحث الثنائي يتم تشغيله Run في وقت لوغاريتمي Logarithmic Time (أو Log Time). فيما يلي جدول يلخص النتائج التي توصلنا إليها اليوم.

أوقات التشغيل Run Times لخوارزميات البحث Search Algorithms

التدوين Big O Notation

تدوين Big O هو تدوين Notation خاص يخبرك بمدى سرعة الخوارزمية. من يهتم؟ حسنًا، منالواضح أنك ستستخدم خوارزميات Algorithms الآخرين كثيرًا - وعندما تفعل ذلك، من الجيد أن تفهم مدى سرعتها Fast أو بطئها Slow. في هذا القسم، سأشرح ماهية تدوين Big O وأعطيك قائمة بأوقات التشغيل الأكثر شيوعًا Running Times للخوارزميات التي تستخدمها.

أوقات تشغيل الخوارزمية Algorithm Running Times تنمو بمعدلات مختلفة

يكتب بوب خوارزمية بحث Search Algorithm لوكالة ناسا NASA. ستنطلق الخوارزمية الخاصة به عندما يكون الصاروخ على وشك الهبوط على القمر، وسوف تساعد في حساب مكان الهبوط.

هذا مثال على كيفية نمو Grow وقت تشغيل Run Time خوارزميتين بمعدلات Rates مختلفة. يحاول بوب Bob الاختيار بين البحث البسيط Simple Search والبحث الثنائي Binary Search. يجب أن تكون الخوارزمية سريعة Fast وصحيحة Correct. من ناحية، يكون البحث الثنائي Binary Search أسرع. ولدى بوب 10 ثوان فقط لمعرفة مكان الهبوط - وإلا فإن الصاروخ سيكون خارج مساره. من ناحية أخرى، البحث البسيط Simple Search أسهل في الكتابة، وهناك فرصة أقل لحدوث أخطاء (الخلل البرمجي) Bugs. وبوب حقًا لا يريد أخطاء برمجية في كود Code لهبوط صاروخ!

لتوخي مزيد من الحذر، قرر بوب ضبط توقيت كلتا الخوارزميتين بقائمة List من 100 عنصر. لنفترض أن التحقق Check من عنصر Element واحد يستغرق 1 مللي ثانية Millisecond. باستخدام البحث البسيط Simple Search، يتعين على بوب التحقق من 100 عنصر، وبالتالي يستغرق البحث 100 مللي ثانية للتشغيل Run. من ناحية أخرى، عليه فقط التحقق من 7 عناصر باستخدام البحث الثنائي Binary Search ($\log_2 100$ يساوي 7 تقريبًا)، بحيث يستغرق البحث 7 مللي ثانية للتشغيل Run. لكن من الناحية الواقعية، ستحتوي القائمة List على أكثر من مليار عنصر. إذا كان الأمر كذلك، فكم من الوقت سيستغرق البحث البسيط Simple Search؟ وكم من الوقت سيستغرق البحث الثنائي Binary Search؟ تأكد من حصولك على إجابة لكل سؤال قبل مواصلة القراءة.

وقت التشغيل للبحث البسيط Simple Search في مقابل البحث الثنائي Binary Search، مع قائمة من 100 عنصر

يقوم بوب بتشغيل بحثًا ثنائيًا Binary Search بمليار عنصر، ويستغرق 30 مللي ثانية ($\log_2 1{,}000{,}000{,}000$ يساوي 30 تقريبًا). "32 مللي ثانية!" هو يفكّر. "البحث الثنائي Binary Search أسرع بنحو 15 مرة من البحث البسيط Simple Search، لأن البحث البسيط استغرق 100 مللي ثانية مع 100 عنصر، والبحث الثنائي استغرق

7 مللي ثانية. لذا فإن البحث البسيط سيستغرق 30 × 15 = 450 مللي ثانية، أليس كذلك؟ أقل بكثير من حدود 10 ثوانٍ. " يقرر بوب أن يستخدم بحث بسيط Simple Search. هل هذا هو الاختيار الصحيح؟

لا، تبين أن بوب مخطئ. مخطئ بالكامل. وقت التشغيل Run Time للبحث البسيط Simple Search الذي يحتوي على مليار عنصر سيكون 1 مليار مللي ثانية، أي 11 يومًا! المشكلة هي أن أوقات تشغيل Run Times البحث الثنائي Binary Search والبحث البسيط Simple Search لا تنمو Grow بنفس المعدل Rate.

عدد العناصر	SIMPLE SEARCH	BINARY SEARCH
100 ELEMENTS	100 ms	7 ms
10,000 ELEMENTS	10 seconds	14 ms
1,000,000,000 ELEMENTS	11 days	32 ms

تنمو أوقات التشغيل Run Times Grow بسرعات مختلفة جدًا!

أي أنه، مع زيادة عدد العناصر، يستغرق البحث الثنائي Binary Search وقتًا أطول قليلاً للتشغيل Run. لكن البحث البسيط Simple Search يستغرق وقتا أكثر بكثير للتشغيل Run. لذلك مع زيادة قائمة الأرقام، يصبح البحث الثنائي Binary Search فجأة أسرع بكثير من البحث البسيط Simple Search. اعتقد بوب أن البحث الثنائي أسرع 15 مرة من البحث البسيط، ولكن هذا ليس صحيحًا. إذا كانت القائمة تحتوي على مليار عنصر، فستكون أسرع بمقدار 33 مليون مرة. لهذا السبب لا يكفي معرفة المدة التي تستغرقها الخوارزمية Algorithm للتشغيل Run - فأنت بحاجة إلى معرفة كيفية زيادة Increase وقت التشغيل Running Time مع زيادة حجم القائمة List Size. وهنا يأتي دور تدوين Big O Notation.

تدوين Big O يخبرك بمدى سرعة الخوارزمية. على سبيل المثال، افترض أن لديك قائمة List حجمها n. يحتاج البحث البسيط Simple Search إلى التحقق Check من كل عنصر Element، لذلك سيستغرق الأمر عدد n من العمليات Operations. وقت التشغيل Run Time في تدوين Big O هو O(n). اين الثواني Seconds؟ لا توجد ثواني — لا يخبرك تدوين Big O بالسرعة في وحدة الثواني. يتيح لك تدوين Big O مقارنة Compare عدد العمليات Operations. يخبرك بمدى سرعة نمو Grow الخوارزمية Algorithm.

فيما يلي مثال آخر. يحتاج البحث الثنائي Binary Search إلى عمليات عددها Log n للتحقق من قائمة List حجمها n. ما هو وقت التشغيل Running Time في تدوين Big O Notation؟ إنه O(log n).

بشكل عام، تدوين Big O يتم كتابته على النحو التالي.

كيف يبدو تدوين Big O Notation

هذا يخبرك بعدد العمليات Operations التي ستقوم بها الخوارزمية Algorithm. يُطلق عليه تدوين Big O لأنك وضعت "O كبيرة" أمام عدد العمليات (تبدو وكأنها مزحة، لكنها صحيحة!).

الآن دعونا نلقي نظرة على بعض الأمثلة. ونري ما إذا كان يمكنك معرفة وقت تشغيل Run Time هذه الخوارزميات Algorithms.

تَصوُّر Visualizing أوقات تشغيل Big O المختلفة

فيما يلي مثال عملي يمكنك اتباعه Follow في المنزل ببعض الورق وقلم رصاص. افترض أن عليك رسم شبكة Grid من 16 مربع Box.

ما هي الخوارزمية Algorithm الجيدة لرسم هذه الشبكة Grid؟

الخوارزمية 1

إحدى الطرق للقيام بذلك هي رسم 16 مربعاً Boxes، واحدًا تلو الآخر. تذكَّر أن تدوين Big O يقوم بحساب عدد العمليات Operations. في هذا المثال، رسم مربع واحد هو عملية واحدة. يجب عليك رسم 16 مربعاً. كم عدد العمليات التي سيستغرقها رسم مربع واحد في كل مرة؟

رسم شبكة Grid مربع واحد في كل مرة

يستغرق رسم 16 مربعاً 16 خطوة. ما هو وقت تشغيل Running Time هذه الخوارزمية Algorithm؟

الخوارزمية 2

قم بتجربة هذه الخوارزمية Algorithm بدلا من ذلك. طيّ الورق Fold The Paper.

في هذا المثال، يعد طيّ الورق مرة واحدة بمثابة عملية Operation. لقد صنعت للتو مربعين Two Boxes بهذه العملية Operation!

اطوِ الورقة مرة أخري ومرة أخري ومرة أخري (3 مرات).

افتح الطيّات بعد أربع طيات، وستحصل على شبكة Grid جميلة! كل طيّة تقوم بمضاعفة Doubles عدد المربعات. لقد صنعت 16 مربعاً بـ 4 عمليات Operations!

رسم شبكة Grid بأربع طيّات Folds

يمكنك صنع ضعف عدد المربعات Boxes مع كل طيّة، إذاً يمكنك رسم 16 مربعا في 4 خطوات. ما هو وقت تشغيل هذه الخوارزمية؟ قم بحساب أوقات التشغيل Running Times لكلتا الخوارزميتين قبل مواصلة القراءة.

الإجابات: تستغرق الخوارزمية 1 وقت $O(n)$، وتستغرق الخوارزمية 2 وقت $O(\log n)$.

يُنشِئ Big O وقت تشغيل أسوأ حالة Worst-Case Running Time

لنفترض أنك تستخدم بحثًا بسيطًا Simple Search للبحث عن شخص في دفتر الهاتف Phone Book. أنت تعلم أن البحث البسيط Simple Search يستغرق وقتًا $O(n)$ للتشغيل Run، مما يعني أنه في أسوأ حالة Worst Case، سيتعين عليك البحث خلال كل إدخال Entry في دفتر هاتفك. في هذه الحالة، أنت تبحث عن Adit. هذا الرجل هو أول إدخال Entry في دفتر هاتفك. لذلك لم يكن عليك إلقاء نظرة على كل إدخال Entry لقد وجدته في المحاولة الأولى. هل استغرقت هذه الخوارزمية وقت $O(n)$؟ أم استغرقت وقت $O(1)$ لأنك وجدت الشخص في المحاولة الأولى؟

لا يزال البحث البسيط Simple Search يأخذ وقت $O(n)$. في هذه الحالة، وجدت ما كنت تبحث عنه على الفور. هذا هو سيناريو أفضل حالة Best-Case Scenario. لكن تدوين Big O يدور حول سيناريو أسوأ حالة Worst-Case Scenario. لذا يمكنك القول أنه، في أسوأ حالة، سيتعين عليك البحث في كل إدخال Entry في دفتر الهاتف مرة واحدة. هذا وقت $O(n)$. وللتأكيد مرة أخري - أنت تعلم أن البحث البسيط Simple Search لن يكون أبدًا أبطأ من وقت $O(n)$.

ملاحظة

إلى جانب وقت التشغيل أسوأ حالة Worst-Case Run Time، من المهم أيضًا إلقاء نظرة على وقت تشغيل الحالة المتوسطة Average-Case Run Time. يتم مناقشة أسوأ حالة Worst Case في مقابل الحالة المتوسطة Average Case في الفصل الرابع.

بعض أوقات تشغيل Big O الشائعة

فيما يلي خمس أوقات تشغيل Big O ستواجهها كثيرًا، مرتبة من الأسرع Fastest إلى الأبطأ Slowest:

- $O(\log n)$ ، المعروف أيضًا باسم الوقت اللوغاريتمي Log Time. مثال: البحث الثنائي Binary Search.
- $O(n)$ ، المعروف أيضًا باسم الوقت الخطي Linear Time. مثال: البحث البسيط Simple Search.
- $O(n * \log n)$ ، مثال: خوارزمية ترتيب سريعة Fast Sorting Algorithm، مثل الترتيب السريع Quicksort (ستظهر في الفصل الرابع).
- $O(n^2)$ ، مثال: خوارزمية ترتيب بطيئة Slow Sorting Algorithm، مثل ترتيب التحديد Selection Sort (سيظهر في الفصل الثاني).
- $O(n!)$ ، مثال: خوارزمية بطيئة حقًا Really Slow Algorithm، مثل مندوب المبيعات المتنقل Traveling Salesperson (التالي!).

لنفترض أنك ترسم شبكة Grid من 16 مربعًا مرة أخرى، ويمكنك الاختيار من بين 5 خوارزميات Algorithms مختلفة للقيام بذلك. إذا استخدمت الخوارزمية الأولى، فسوف يستغرق الأمر وقت $O(\log n)$ لرسم الشبكة.

يمكنك إجراء 10 عمليات Operations في الثانية. مع وقت $O(\log n)$، سوف يستغرق الأمر 4 عمليات لرسم شبكة من 16 مربعًا (log16 هو 4). لذلك سوف يستغرق الأمر 0.4 ثانية لرسم الشبكة. ماذا لو كان عليك رسم 1024 مربعا؟ سوف يستغرق الأمر log1024 = 10 عمليات، أو ثانية واحدة لرسم شبكة من 1024 مربع. هذه الأرقام باستخدام الخوارزمية الأولى.

الخوارزمية الثانية أبطأ: تستغرق وقت $O(n)$. ستستغرق رسم 16 مربعاً 16 عملية، وسيستغرق رسم 1024 مربعا 1024 عملية. كم هذا الوقت بالثواني؟

فيما يلي المدة التي سيستغرقها رسم شبكة Grid لبقية الخوارزميات Algorithms، من الأسرع إلى الأبطأ:

هناك أوقات تشغيل Run Times أخرى أيضًا، ولكن هذه هي الخمس مرات الأكثر شيوعًا.

هذا تبسيط. في الواقع، لا يمكنك التحويل من وقت تشغيل Big O إلى عدد من العمليات Operations بدقة، ولكن هذا جيد بما يكفي في الوقت الحالي. سنعود إلى تدوين Big O في الفصل الرابع، بعد أن تتعلم المزيد من الخوارزميات Algorithms. في الوقت الحالي، النقاط الرئيسية هي كما يلي:

- لا تُقاس سرعة الخوارزمية Algorithm Speed بالثواني Seconds، بل تقاس بنمو Growth عدد العمليات Operations.
- بدلاً من ذلك، نتحدث عن مدى سرعة زيادة وقت تشغيل Run Time الخوارزمية Algorithm مع زيادة حجم الإدخال Input Size.
- يتم التعبير عن وقت تشغيل الخوارزميات Algorithm Run Time بواسطة تدوين Big O Notation.
- $O(\log n)$ أسرع من $O(n)$، لكنها تزداد سرعة مع نمو قائمة العناصر التي تبحثها.

التمارين Exercises

اذكر وقت التشغيل Run Time لكل من هذه السيناريوهات Scenarios من حيث تدوين Big O.

1.3 لديك اسم Name وتريد العثور على رقم هاتف الشخص في دفتر الهاتف.

1.4 لديك رقم هاتف وتريد العثور على اسم الشخص في دفتر الهاتف. (تلميح: عليك البحث خلال الكتاب بأكمله!)

1.5 تريد قراءة أرقام كل شخص في دليل الهاتف.

1.6 تريد قراءة أرقام للأسماء بحرف A فقط. (هذا سؤال محيرا! إنه يتضمن مفاهيم تم تناولها أكثر في الفصل الرابع. اقرأ الإجابة - قد تتفاجأ!)

مندوب المبيعات المتنقل Traveling Salesperson

ربما تكون قد قرأت القسم الأخير وتفكر، "لا يمكن بأي حال من الأحوال أن أواجه خوارزمية Algorithm تستغرق وقتًا $O(n!)$." حسنًا، دعني أحاول إثبات أنك مخطئ! فيما يلي مثال على خوارزمية Algorithm ذات وقت تشغيل Running Time سيئ حقًا. هذه مسألة Problem مشهورة في علوم الكمبيوتر Computer Science، لأن نموها Growth مروع ويعتقد بعض الأشخاص الأذكياء جدًا أنه لا يمكن تحسينها. إنها تسمى مسألة مندوب المبيعات المتنقل Traveling Salesperson Problem.

لديك مندوب مبيعات Salesperson.

مندوب المبيعات يجب أن يذهب إلى خمس مدن.

يريد مندوب المبيعات هذا، الذي سأطلق عليه اسم Opus، الوصول إلى جميع المدن الخمس أثناء قطع أقل المسافة Minimum Distance. وإليك طريقة واحدة للقيام بذلك: انظر إلى كل ترتيب Order محتمل يمكن من خلاله السفر إلى المدن.

يقوم بجمع المسافة الإجمالية Total Distance ثم يختار المسار Path الأقل مسافة. يوجد 120 تباديل Permutations مع 5 مدن، لذلك سوف يستغرق الأمر 120 عملية Operation لحل المسألة Problem لـ 5 مدن. بالنسبة ل 6 مدن، سوف يستغرق الأمر 720 عملية (هناك 720 تبديل Permutation). بالنسبة لـ 7 مدن، سيستغرق الأمر 5،040 عملية!

عدد العمليات Number Of Operations يزيد Increases بشكل جذري Drastically

بشكل عام، بالنسبة لعدد n من العناصر Items، سوف يستغرق الأمر وقت !n (مضروب n) (n Factorial) لحساب النتيجة. إذن هذا هو وقت O(n!)، أو وقت المضروب Factorial Time. يتطلب الأمر الكثير من العمليات Operations لكل شيء باستثناء أصغر الأرقام. بمجرد أن تتعامل مع أكثر من 100 مدينة، من المستحيل حساب الإجابة في الوقت المناسب - سوف تنهار الشمس أولاً.

هذه خوارزمية رهيبة! يجب أن يستخدم Opus خوارزمية مختلفة، أليس كذلك؟ لكنه لا يستطيع. هذه إحدى المسائل التي لم يتم حلها Unsolved Problems في علوم الحاسب Computer Science.

لا توجد خوارزمية Algorithm معروفة سريعة لذلك، ويعتقد الأشخاص الأذكياء أنه من المستحيل وجود خوارزمية ذكية Smart Algorithm لهذه المسألة Problem. أفضل ما يمكننا فعله هو التوصل إلى حل تقريبي Approximate Solution؛ انظر الفصل العاشر للمزيد.

ملاحظة أخيرة: إذا كنت قارئًا متقدمًا Advanced Reader، فقم بتصفح أشجار البحث الثنائية Binary Search Trees! يوجد وصف موجز لها في الفصل الأخير.

الخلاصة

- البحث الثنائي Binary Search أسرع بكثير من البحث البسيط Simple Search.
- $O(\log n)$ أسرع من $O(n)$، لكنها تصبح أسرع كثيرًا بمجرد نمو قائمة العناصر التي تبحث خلالها.
- لا تقاس سرعة الخوارزمية Algorithm Speed بالثواني.
- يتم قياس أوقات الخوارزمية Algorithm Times من حيث نمو الخوارزمية Algorithm Growth.
- تتم كتابة أوقات الخوارزمية بواسطة تدوين Big O Notation.

Coding كوي

2 ترتيب التحديد
Selection Sort

في هذا الفصل

- تتعرف على المصفوفات Arrays والقوائم المرتبطة Linked Lists - وهما اثنان من هياكل البيانات الأساسية Basic Data Structures. والتي يتم استخدامها قطعاً في كل مكان. لقد استخدمت بالفعل المصفوفات Arrays في الفصل الأول، وسوف تستخدمها في كل فصل تقريبًا من هذا الكتاب.
المصفوفات موضوع بالغ الأهمية، لذا انتبه! لكن في بعض الأحيان يكون من الأفضل استخدام قائمة مرتبطة Linked List بدلاً من المصفوفة Array. يشرح هذا الفصل إيجابيات Pros وسلبيات Cons كليهما حتى تتمكن من تحديد أيهما مناسب للخوارزمية Algorithm الخاصة بك.
- تتعلم خوارزمية الترتيب Sorting Algorithm الأولي لك. تنجح الكثير من الخوارزميات Algorithms فقط إذا تم ترتيب Sort البيانات Data. تذكر البحث الثنائي Binary Search؟ يمكنك إجراء بحث ثنائي فقط على قائمة مرتبة Sorted List من العناصر Elements. هذا الفصل يعلمك ترتيب التحديد Selection Sort. معظم اللغات Languages لديها خوارزمية ترتيب Sorting Algorithm مدمجة Built-In، لذلك نادرًا ما سوف تحتاج إلى كتابة إصدارك الخاص من البداية.
لكن ترتيب التحديد Selection Sort هو نقطة انطلاق للترتيب السريع Quicksort، والذي سأتناوله في الفصل التالي. الترتيب السريع Quicksort هو خوارزمية مهمة، وسيكون من الأسهل فهمها إذا كنت تعرف خوارزمية ترتيب Sorting Algorithm واحدة بالفعل.

Coding عربي

ما تحتاج إلى معرفته

لفهم تفاصيل تحليل الأداء Performance Analysis في هذا الفصل، تحتاج إلى معرفة Big O Notation واللوغاريتمات Logarithms. إذا كنت لا تعرفهم، أقترح عليك العودة وقراءة الفصل الأول. سيتم استخدام تدوين Big O Notation في بقية الكتاب.

كيف تعمل الذاكرة Memory

تخيل أنك تذهب إلى عرض Show وتحتاج إلى التحقق من الأشياء الخاصة بك. متاح لك خزانة ذات أدراج.

يمكن لكل درج استيعاب عنصر Element واحد. أنت تريد تخزين شيئين، لذلك تقوم بطلب درجين.

أنت تقوم بتخزين Store شيئين هنا.

وأنت الآن على استعداد للعرض! هذه هي الطريقة التي تعمل بها ذاكرة جهاز الكمبيوتر Computer Memory الخاص بك. يبدو جهاز الكمبيوتر الخاص بك كمجموعة ضخمة من الأدراج، ولكل درج عنوان Address.

fe0ffeeb هو عنوان Address خانة Slot في الذاكرة Memory.

في كل مرة تريد تخزين Store عنصر Item في الذاكرة Memory، تطلب من الكمبيوتر بعض المساحة Space، ويعطيك عنوانًا Address حيث يمكنك تخزين العنصر الخاص بك. إذا كنت تريد تخزين عناصر متعددة Multiple Items، فهناك طريقتان أساسيتان للقيام بذلك: المصفوفات Arrays والقوائم Lists. سأتحدث عن المصفوفات والقوائم بعد ذلك، بالإضافة إلى إيجابيات وسلبيات كل منها. لا توجد طريقة واحدة صحيحة لتخزين العناصر لكل حالة استخدام Use Case، لذلك من المهم معرفة الاختلافات.

المصفوفات Arrays والقوائم المرتبطة Linked Lists

تحتاج أحيانًا إلى تخزين قائمة بالعناصر في الذاكرة. لنفترض أنك تكتب تطبيقًا App لإدارة مهامك To-Do. ستحتاج إلى تخزين المهام كقائمة List في الذاكرة Memory.

هل يجب استخدام مصفوفة Array أم قائمة مرتبطة Linked List؟ دعنا نقوم أولاً بتخزين المهام في مصفوفة، لأنها أسهل في الاستيعاب. استخدام مصفوفة يعني أن جميع مهامك يتم تخزينها بشكل متجاور Contiguously (بجوار بعضها البعض) في الذاكرة Memory.

افترض الآن أنك تريد إضافة Add مهمة رابعة. ولكن الدرج التالي تم شغله بواسطة متعلقات شخص آخر!

يشبه الأمر الذهاب إلى فيلم مع أصدقائك والعثور على مكان للجلوس - ينضم إليك صديق آخر، ولا يوجد مكان له. عليك أن تنتقل إلى مكان جديد يستوعبكما معاً. في هذه الحالة، تحتاج إلى أن تطلب من جهاز الكمبيوتر الخاص بك مساحة مختلفة من الذاكرة يمكن أن تتسع لأربع مهام. ثم تحتاج إلى نقل جميع المهام الخاصة بك إلي هناك.

إذا جاء صديق آخر، فليس لديك مساحة مرة أخرى - وعليكم جميعًا الانتقال مرة أخرى! يا له من أمر مزعج. وبالمثل، فإن إضافة Add عناصر جديدة New Items إلى مصفوفة Array يمكن أن يكون مصدر إزعاج كبير. إذا نفدت المساحة وتحتاج إلى الانتقال إلى موقع جديد في الذاكرة Memory في كل مرة، فستكون إضافة عنصر جديد بطيئة حقًا.

أحد الحلول السهلة هو "الاحتفاظ بالمقاعد": حتى إذا كان لديك 3 عناصر فقط في قائمة المهام، يمكنك أن تطلب من الكمبيوتر 10 خانات Slots، بشكل احتياطي فقط. ثم يمكنك إضافة 10 عناصر إلى قائمة المهام الخاصة بك دون الحاجة إلى الانتقال. هذا حل جيد، لكن يجب أن تكون على دراية ببعض الجوانب السلبية:

- قد لا تحتاج إلى الخانات Slots الإضافية التي طلبتها، وهكذا سيتم إهدار تلك الذاكرة Memory. أنت لا تستخدمها، ولا يمكن أيضًا لأي شخص آخر استخدامها.
- قد تحتاج إلى إضافة أكثر من 10 عناصر إلى قائمة المهام الخاصة بك ويجب عليك حينها الانتقال على أي حال.

لذلك فهو حل بديل جيد، لكنه ليس حلاً مثاليًا. القوائم المرتبطة Linked Lists تقوم بحل مسألة إضافة Add العناصر Items.

مع القوائم المرتبطة، يمكن أن تكون عناصرك Items في أي مكان في الذاكرة Memory.

كل عنصر يقوم بتخزين عنوان Address العنصر التالي في القائمة List. يتم ربط Link مجموعة من عناوين الذاكرة العشوائية Random Memory Addresses ببعضها البعض.

عناوين الذاكرة المرتبطة Linked Memory Addresses

إنه مثل البحث عن الكنز. تذهب إلى العنوان Address الأول، ويقول، "يمكن العثور على العنصر Item التالي في العنوان 123" لذلك تذهب إلى العنوان 123، ويقول، "يمكن العثور على العنصر التالي في العنوان 847" وهكذا. من السهل إضافة عنصر إلى قائمة مرتبطة Linked List: يمكنك وضعه في أي مكان في الذاكرة وتخزين العنوان مع العنصر السابق.

مع القوائم المرتبطة Linked Lists، لن تضطر أبدًا إلى نقل Move العناصر الخاصة بك. أنت أيضا تتجنب مشكلة أخرى. لنفترض أنك ذهبت إلى فيلم شهير مع خمسة من أصدقائك. ستة منكم يحاولون إيجاد مكان

للجلوس، لكن المسرح ممتلئ. لا توجد ستة مقاعد معًا. حسنًا، يحدث هذا أحيانًا مع المصفوفات Arrays.
لنفترض أنك تحاول العثور على 10000 خانة Slot لمصفوفة Array. تحتوي ذاكرتك Memory على 10000 خانة، لكنها لا تحتوي على 10000 خانة معًا. لا يمكنك الحصول على مساحة Space للمصفوفة الخاصة بك!
القائمة المرتبطة Linked List مثل قول "دعونا نفترق ونشاهد الفيلم". إذا كانت هناك مساحة في الذاكرة، فلديك مساحة لقائمتك المرتبطة.
إذا كانت القوائم المرتبطة Linked Lists أفضل بكثير في الإدراجات Inserts، فما فائدة المصفوفات Arrays؟

المصفوفات Arrays

مواقع الويب Websites التي تحتوي على قوائم أفضل 10 (Top-10 Lists) تستخدم أسلوبًا وهميًا للحصول على المزيد من مشاهدات الصفحات Page Views. بدلاً من إظهار القائمة لك في صفحة واحدة، يقومون بوضع عنصرًا Item واحدًا في كل صفحة ويجعلونك تنقر Click فوق زر التالي Next Button للوصول إلى العنصر التالي في القائمة. على سبيل المثال، أفضل 10 أشرار تلفزيونيين (Top 10 Best TV Villains) لن يقوم بإظهار القائمة الكاملة لك في صفحة واحدة. بدلاً من ذلك، تبدأ من رقم 10 (Newman)، وعليك النقر فوق Next في كل صفحة للوصول إلى رقم 1 (Gustavo Fring). هذا الأسلوب يمنح مواقع الويب 10 صفحات كاملة لعرض الإعلانات عليها، ولكن من الممل النقر على Next تسع مرات للوصول إلى رقم 1. سيكون من الأفضل بكثير إذا كانت القائمة بأكملها في صفحة واحدة ويمكنك النقر فوق اسم كل شخص لمزيد من المعلومات.

القوائم المرتبطة Linked Lists لديها مشكلة مماثلة. افترض أنك تريد قراءة العنصر الأخير Last Item في قائمة مرتبطة. لا يمكنك قراءته Read فحسب، لأنك لا تعرف العنوان Address الموجود فيه. بدلاً من ذلك، يجب عليك الانتقال إلى العنصر Item رقم 1 للحصول على عنوان Address العنصر رقم 2. ثم يتعين عليك الانتقال إلى العنصر رقم 2 للحصول على عنوان العنصر رقم 3. وهكذا، حتى تصل إلى العنصر الأخير.
القوائم المرتبطة Linked Lists تكون رائعة إذا كنت ستقرأ Read جميع العناصر واحدًا تلو الآخر: يمكنك قراءة عنصر واحد، واتباع Follow العنوان إلى العنصر التالي، وهكذا. ولكن إذا كنت ستستمر في القفز، فإن القوائم المرتبطة Linked Lists ستكون سيئة.

المصفوفات Arrays مختلفة. أنت تعرف عنوان Address كل عنصر Item في مصفوفتك. على سبيل المثال، افترض أن المصفوفة الخاصة بك تحتوي على خمسة عناصر، وأنت تعلم أنها تبدأ من العنوان 00. ما هو عنوان العنصر رقم 5؟

تخبرك الرياضيات البسيطة Simple Math: أن العنوان يكون 04.

المصفوفات Arrays تكون رائعة إذا كنت تريد قراءة عناصر عشوائية Random Elements، لأنه يمكنك البحث بشكل فوري Instantly عن أي عنصر في المصفوفة. مع القائمة المرتبطة Linked List، لا تكون العناصر بجوار بعضها البعض، لذلك لا يمكنك حساب موضع Position العنصر الخامس في الذاكرة Memory على الفور - عليك الانتقال إلى العنصر الأول للحصول على عنوان العنصر الثاني، ثم الانتقال إلى العنصر الثاني للحصول على عنوان العنصر الثالث، وهكذا حتى تصل إلى العنصر الخامس.

المصطلحات Terminology

يتم ترقيم العناصر Elements الموجودة في المصفوفة Array. يبدأ هذا الترقيم Numbering من 0، وليس 1. على سبيل المثال، في هذه المصفوفة Array، الرقم 20 يكون في الموضع 1.

والرقم 10 يكون في الموضع 0. هذا عادة ما يجعل المبرمجين الجدد يدورون حول أنفسهم. البدء من 0 يجعل كتابة جميع أنواع الأكواد القائمة على المصفوفات Array-Based Code أسهل، لذلك تمسَّك المبرمجون بها.

تقريبًا كل لغة برمجة Programming Language تستخدمها ستقوم بترقيم Numbering عناصر المصفوفة Array Elements بدءًا من 0. وستعتاد عليها قريبًا.

موضع العنصر Element Position يسمى بالفهرس (الرقم التسلسلي) Index.

لذا فبدلاً من القول، "الرقم 20 في الموضع 1"، المصطلح الصحيح هو "الرقم 20 في الفهرس 1"

سأقوم باستخدام مصطلح الفهرس Index للإشارة إلى الموضع Position في هذا الكتاب.

فيما يلي أوقات التشغيل Run Times للعمليات الشائعة Common Operations على المصفوفات Arrays والقوائم Lists.

		المصفوفات ARRAYS	القوائم LISTS
القراءة	READING	O(1)	O(n)
الإدراج	INSERTION	O(n)	O(1)

O(n) = LINEAR TIME الوقت الخطي
O(1) = CONSTANT TIME الوقت الثابت

O(1) تمثل الوقت الثابت Constant Time.

سؤال: لماذا يستغرق إدراج عنصر Insert Element في مصفوفة Array الوقت O(n)؟ افترض أنك تريد إدراج عنصر في بداية المصفوفة. كيف يمكنك أن تفعل هذا؟ كم من الوقت سوف يستغرق؟ اعثر على إجابات لهذه الأسئلة في القسم التالي!

التمرين Exercise

2.1 لنفترض أنك تقوم ببناء تطبيق App لتتبع Track مواردك المالية Finances.

1. البقالة
2. الأفلام
3. اشتراك عضوية

كل يوم، تكتب كل شيء أنفقت عليه المال. في نهاية الشهر، تقوم بمراجعة نفقاتك وتقوم بجمع إجمالي المبلغ الذي أنفقته. إذاً، لديك الكثير من الإدخالات (الإدراجات) Inserts والقليل من القراءات Reads.

هل يجب عليك استخدام مصفوفة Array أم قائمة List؟

الإدراج Inserting في منتصف القائمة List

افترض أنك تريد أن تعمل قائمة المهام To-Do List الخاصة بك مثل التقويم Calendar. في وقت سابق، كنت تقوم بإضافة Add أشياء إلى نهاية القائمة.

الآن تريد إضافتها بالترتيب Order الذي ينبغي إجراؤها به.

مرتبة (Ordered) غير مرتبة (Unordered)

ما الأفضل إذا كنت تريد إدراج Insert عناصر Elements في الوسط: المصفوفات Arrays أم القوائم Lists؟

مع القوائم Lists، يكون الأمر سهلاً مثل تغيير ما يشير إليه العنصر السابق.

ولكن بالنسبة للمصفوفات Arrays، يجب عليك نقل Shift جميع العناصر المتبقية إلى أسفل.

وإذا لم تكن هناك مساحة Space، فقد تضطر إلى نسخ Copy كل شيء إلى موقع Location جديد!

القوائم Lists أفضل إذا كنت تريد إدراج عناصر Elements في المنتصف Middle.

عمليات الحذف Deletions

ماذا لو كنت تريد حذف Delete عنصر Element؟ مرة أخرى، القوائم Lists أفضل، لأنك تحتاج فقط إلى تغيير ما يشير إليه العنصر السابق. باستخدام المصفوفات Arrays، يجب نقل كل شيء لأعلى عند حذف عنصر.

على عكس عمليات الإدراج (إضافة العناصر) Insertions، ستنجح دائمًا عمليات الحذف Deletions. يمكن أن تفشل عمليات الإدراج في بعض الأحيان عند عدم وجود مساحة Space في الذاكرة Memory. ولكن يمكنك دائمًا حذف عنصر. فيما يلي أوقات التشغيل Run Times للعمليات الشائعة Common Operations على المصفوفات Arrays والقوائم المرتبطة Linked Lists.

		المصفوفات ARRAYS	القوائم LISTS	
القراءة	READING	O(1)	O(n)	وقت التشغيل
الإدراج	INSERTION	O(n)	O(1)	
الحذف	DELETION	O(n)	O(1)	

من الجدير بالذكر أن عمليات الإدراج Insertions والحذف Deletions تستغرق وقت O(1) فقط إذا كان بإمكانك الوصول Access بشكل فوري Instantly إلى العنصر المراد حذفه. من الممارسات الشائعة القيام بتتبع Track العناصر الأولى والأخيرة في القائمة المرتبطة Linked List، لذلك لن يستغرق حذفها سوى وقت O(1).

أيهما يستخدم أكثر: المصفوفات Arrays أم القوائم Lists؟ من الواضح أن ذلك يعتمد على حالة الاستخدام Use Case. لكن المصفوفات تشهد استخداماً كبيراً لأنها تسمح بالوصول العشوائي Random Access. هناك نوعان مختلفان من الوصول: الوصول العشوائي Random Access والوصول التسلسلي Sequential Access. الوصول التسلسلي Sequential Access يعني قراءة Reading العناصر Elements واحداً تلو الآخر، بدءًا من العنصر الأول. القوائم المرتبطة Linked Lists يمكنها فقط القيام بوصول تسلسلي Sequential Access. إذا كنت ترغب في قراءة العنصر العاشر من قائمة مرتبطة، يجب عليك قراءة العناصر التسعة الأولى واتباع Follow الروابط Links إلى العنصر العاشر. الوصول العشوائي Random Access يعني أنه يمكنك القفز مباشرة إلى العنصر العاشر. ستسمعني كثيرًا أقول إن المصفوفات Arrays أسرع في عمليات القراءة Reads. هذا لأنها توفر وصول عشوائي Random Access. تتطلب الكثير من حالات الاستخدام Use Cases الوصول العشوائي، لذلك يتم استخدام المصفوفات كثيرًا. المصفوفات Arrays والقوائم Lists يتم استخدامها أيضًا لتنفيذ Implement هياكل بيانات الأخرى Data Structures (ستظهر لاحقًا في الكتاب).

2.2 لنفترض أنك تنشئ تطبيقًا App للمطاعم لتلقي طلبات العملاء. يحتاج تطبيقك إلى تخزين قائمة الطلبات. يستمر النادل Waiter في إضافة الطلبات إلى هذه القائمة List، ويقوم الطاهي Chef بسحب الطلبات من القائمة وطبخها. إنها قائمة انتظار للطلبات Order Queue: يقوم النادل بإضافة الطلبات إلى أسفل قائمة الانتظار Queue، ويسحب الطاهي الطلب الأول من أعلي قائمة الانتظار Queue.

هل ستستخدم مصفوفة Array أو قائمة مرتبطة Linked List لتنفيذ Implement قائمة الانتظار Queue هذه؟ (تلميح: القوائم المرتبطة Linked Lists جيدة لعمليات الإدخال Inserts/الحذف Deletes، والمصفوفات Arrays جيدة للوصول العشوائي Random Access. ما هو اختيارك الذي سوف تستخدمه في هذا التطبيق؟)

2.3 دعونا نجري تجربة فكرية. افترض أن موقع فيسبوك Facebook يحتفظ بقائمة List بأسماء المستخدمين Usernames. عندما يحاول شخص ما تسجيل الدخول Log In إلى Facebook، يتم إجراء بحث Search عن اسم المستخدم الخاص به. إذا كان اسمه مدرجًا في قائمة أسماء المستخدمين، فيمكنه تسجيل الدخول. يسجل الأشخاص الدخول إلى Facebook في كثير من الأحيان، لذلك هناك الكثير من عمليات البحث خلال قائمة أسماء المستخدمين هذه. افترض أن Facebook يستخدم بحثًا ثنائيًا Binary Search للبحث خلال القائمة. يحتاج البحث الثنائي إلى وصول عشوائي Random Access - يجب أن تكون قادرًا على الوصول إلى منتصف قائمة أسماء المستخدمين على الفور Instantly. مع العلم بهذا، هل ستقوم بتنفيذ Implement القائمة كمصفوفة أم قائمة مرتبطة؟

2.4 يشترك Sign Up الأشخاص في Facebook في كثير من الأحيان أيضًا. لنفترض أنك قررت استخدام مصفوفة Array لتخزين قائمة المستخدمين. ما هي سلبيات المصفوفة Array للإدخالات Inserts؟ على وجه الخصوص، افترض أنك تستخدم البحث الثنائي Binary Search للبحث خلال عمليات تسجيل الدخول Logins. ماذا يحدث عند إضافة Add مستخدمين جدد إلى مصفوفة Array؟

2.5 في الواقع، لا يستخدم Facebook مصفوفة Array أو قائمة مرتبطة Linked List لتخزين معلومات المستخدم User Information. لنفكر في هيكل بيانات مختلط Hybrid Data Structure: مصفوفة من القوائم المرتبطة Array Of Linked Lists. لديك مصفوفة بها 26 خانة Slot. تشير كل خانة إلى قائمة مرتبطة Linked List. على سبيل المثال، تشير الخانة الأولى في المصفوفة إلى قائمة مرتبطة تحتوي على جميع أسماء المستخدمين التي تبدأ بحرف a. تشير الخانة الثانية إلى قائمة مرتبطة تحتوي على جميع أسماء المستخدمين التي تبدأ بحرف b، وهكذا.

افترض أن Adit B يقوم بالاشتراك Signs Up في Facebook، وتريد إضافته إلى القائمة. تذهب إلى الخانة 1 في المصفوفة، وتنتقل إلى القائمة المرتبطة للخانة 1، وتقوم بإضافة Adit B في النهاية. الآن، لنفترض أنك تريد البحث عن Zakhir H. تذهب إلى الخانة 26، والتي تشير إلى قائمة مرتبطة Linked List لكل الأسماء التي تبدأ بحرف Z. ثم تبحث خلال تلك القائمة لتجد Zakhir H.

قارن هيكل البيانات المختلط- الهجين Hybrid Data Structure هذا بالمصفوفات Arrays والقوائم المرتبطة Linked Lists. هل هو أبطأ أم أسرع من كل منهما للبحث Searching والإدخال Inserting؟ لا يتعين عليك تحديد أوقات تشغيل Big O، فقط ما إذا كان هيكل البيانات الجديد سيكون أسرع أم أبطأ.

ترتيب التحديد Selection Sort

دعونا نجمع كل ذلك معًا لنتعلم الخوارزمية الثانية: ترتيب التحديد Selection Sort. لمتابعة هذا القسم، تحتاج إلى فهم المصفوفات Arrays والقوائم Lists، بالإضافة إلى تدوين Big O Notation.

افترض أن لديك مجموعة من ملفات الموسيقى على جهاز الكمبيوتر الخاص بك. لكل فنان، لديك عدد مرات التشغيل التي قمت بها.

تريد ترتيب Sort هذه القائمة List من الأكثر إلى الأقل تشغيلًا، بحيث يمكنك ترتيب الفنانين المفضلين لديك.

كيف يمكنك أن تفعل ذلك؟

إحدى الطرق هي تصفح القائمة والعثور على الفنان الأكثر تشغيلاً. ثم تقوم بإضافة هذا الفنان إلى قائمة جديدة New List.

افعل ذلك مرة أخرى للعثور على الفنان التالي الأكثر تشغيلاً.

استمر في فعل ذلك، وسوف ينتهي بك الأمر بقائمة مرتبة Sorted List.

دعونا نرتدي قبعات علوم الكمبيوتر Computer Science الخاصة بنا ونرى كم سيستغرق ذلك من الوقت للتشغيل Run. تذكّر أن الوقت $O(n)$ يعني أنك تمر على كل عنصر في القائمة مرة واحدة. على سبيل المثال، تشغيل Run بحث بسيط Simple Search علي قائمة الفنانين يعني المرور علي كل فنان مرة واحدة.

للعثور على الفنان صاحب أعلى عدد مرات تشغيل، يجب عليك التحقق Check من كل عنصر Item في القائمة list. هذا يستغرق وقت $O(n)$، كما رأيت للتو. إذن لديك عملية Operation تستغرق وقتًا $O(n)$، وعليك القيام بذلك عدد n من المرات:

هذا يستغرق وقت $O(n \times n)$ أو وقت $O(n^2)$.

خوارزميات الترتيب Sorting Algorithms مفيدة جدا. الآن يمكنك ترتيب

- الأسماء في دفتر الهاتف
- مواعيد السفر
- رسائل البريد الإلكتروني Emails (من الأحدث Newest إلى الأقدم Oldest)

التحقق Checking من عدد أقل من العناصر Elements في كل مرة

ربما تتساءل: مع تقدمك في العمليات Operations، يتناقص باستمرار عدد العناصر التي عليك التحقق منها. في النهاية، يتعين عليك التحقق من عنصر واحد فقط. فكيف يمكن أن يظل وقت التشغيل $O(n^2)$؟ هذا سؤال جيد، والإجابة لها علاقة بالثوابت Constants في تدوين Big O Notation. سوف أتطرق إلى هذا أكثر في الفصل الرابع، ولكن إليك الخلاصة.

أنت محق في أنك لست مضطرًا إلى التحقق من قائمة عناصر عددها n في كل مرة. تقوم بالتحقق من عدد n من العناصر، ثم n - 1 ، n - 2 ... 2 ، 1. في المتوسط، تقوم بالتحقق من قائمة تحتوي على $1/2 \times n$ من العناصر. وقت التشغيل هو $O(n \times 1/2 \times n)$. لكن يتم تجاهل الثوابت Constants مثل 1/2 في تدوين Big O Notation (مرة أخرى، انتظر الفصل الرابع للمناقشة الكاملة)، لذلك تكتب $O(n \times n)$ أو $O(n^2)$.

ترتيب التحديد Selection Sort هو عبارة عن خوارزمية رائعة، ولكنها ليست سريعة جدًا. الترتيب السريع Quicksort هي خوارزمية ترتيب أسرع لا تستغرق سوى وقت $O(n \log n)$. إنها قادمة في الفصل التالي!

قائمة أمثلة الكود Example Code Listing

لم نعرض لك الكود Code الخاص بترتيب Sort قائمة الموسيقى، ولكن فيما يلي بعض الكود Code الذي سيفعل شيئًا مشابهًا جدًا: ترتيب Sort مصفوفة Array من الأصغر إلى الأكبر. هيا نقوم بكتابة Write دالة Function للعثور على أصغر عنصر Smallest Element في المصفوفة:

```
def findSmallest(arr):
  smallest = arr[0]
  smallest_index = 0
  for i in range(1, len(arr)):
    if arr[i] < smallest:
      smallest = arr[i]
      smallest_index = i
  return smallest_index
```

يقوم بتخزين أقل قيمة

يقوم بتخزين فهرس Index أقل قيمة

الآن يمكنك استخدام هذه الدالة Function لكتابة ترتيب التحديد Selection Sort:

```
def selectionSort(arr):
  newArr = []
  for i in range(len(arr)):
    smallest = findSmallest(arr)
    newArr.append(arr.pop(smallest))
  return newArr

print selectionSort([5, 3, 6, 2, 10])
```

يقوم بترتيب مصفوفة

يعثر علي أصغر عنصر في المصفوفة ويقوم بإضافته إلى المصفوفة الجديدة

الخلاصة

- ذاكرة جهاز الكمبيوتر Computer Memory مثل مجموعة ضخمة من الأدراج.
- عندما تريد تخزين Store عناصر متعددة Multiple Elements، قم باستخدام مصفوفة Array أو قائمة List.
- باستخدام المصفوفة Array، يتم تخزين جميع العناصر Elements بجوار بعضها البعض.
- باستخدام القائمة List، تتناثر العناصر في كل مكان، ويقوم كل عنصر بتخزين عنوان Address العنصر التالي.
- المصفوفات Arrays تسمح بعمليات قراءة سريعة Fast Reads.
- القوائم المرتبطة Linked Lists تسمح بعمليات إدراج Inserts وحذف Deletes سريعة.
- يجب أن تكون جميع العناصر Elements في المصفوفة Array من نفس النوع Same Type (إما أعداد صحيحة Ints، أو أعداد عشرية Doubles، وهكذا).

Coding Sky

3 التكـــرارية
Recursion

في هذا الفصل

- تتعلم عن التكرارية Recursion. التكرارية هي أسلوب برمجة Coding مستخدم في العديد من الخوارزميات Algorithms. إنه لبنة أساسية Building Block لفهم الفصول اللاحقة في هذا الكتاب.
- تتعلم كيفية تقسيم المسألة Problem إلى حالة أساسية Base Case وحالة تكرارية Recursive Case. إستراتيجية فرِّق تَسُد Divide-And-Conquer Strategy (الفصل الرابع) تستخدم هذا المفهوم البسيط لحل المسائل الصعبة.

أنا متحمس بشأن هذا الفصل لأنه يغطي التكرارية Recursion، وهي طريقة أنيقة Elegant لحل المسائل Solve Problems. التكرارية Recursion هو أحد موضوعاتي المفضلة، لكنه مثير للخلاف. الناس إما يحبونه أو يكرهونه، أو يكرهونه حتى يتعلموا حبه بعد سنوات قليلة. أنا شخصياً كنت في ذلك المعسكر الثالث. لتسهيل الأمور عليك، لدي بعض النصائح:

- هذا الفصل يحتوي على الكثير من أمثلة الكود Code Examples. قم بتشغيل Run الكود Code بنفسك لترى كيف يعمل.
- سوف أتحدث عن الدوال التكرارية Recursive Functions. قم بتجربة دالة تكرارية Recursive Function باستخدام القلم والورق مرة واحدة على الأقل:

مثل، "أن تقوم بتمرير الرقم 5 إلى دالة المضروب Factorial (حاصل ضرب أي عدد صحيح Integer مع كل الأرقام الصحيحة Integers الأقل منه – بمعني ان مضروب 5 (1*2*3*4*5) يساوي 120)، ثم تقوم بالإرجاع Return لهذه الدالة 5 مرات بتمرير الأرقام من 4 إلى 1 إلى دالة المضروب Factorial... وهكذا". العمل خلال دالة Function مثل هذه سوف يعلمك كيفية عمل الدالة التكرارية Recursive Function.

يتضمن هذا الفصل أيضًا الكثير من الكود الزائف Pseudocode. الكود الزائف هو وصف عالي المستوى High-Level Description للمسألة Problem التي تحاول حلها بالكود. إنه يتم كتابته مثل الكود Code، لكن من المفترض أن يكون أقرب إلى كلام البشر.

التكرارية Recursion

لنفترض أنك تحفر في أرض جَدّتك ووجدت حقيبة سفر غامضة مغلقة.

تخبرك جدتك أن مفتاح الحقيبة ربما يكون في هذا الصندوق Box الآخر.

يحتوي هذا الصندوق على المزيد من الصناديق، مع وجود المزيد من الصناديق داخل تلك الصناديق. المفتاح موجود في صندوق في مكان ما. ما هي الخوارزمية Algorithm الخاصة بك للبحث عن المفتاح؟ فكر في خوارزمية قبل أن تستمر في القراءة.

فيما يلي إحدى الطرق.

1. اصنع كومة من الصناديق للبحث خلالها.
2. اسحب صندوق، وانظر داخله.
3. إذا وجدت صندوقًا، قم بإضافته إلى الكومة للنظر فيه لاحقًا.
4. إذا وجدت مفتاحًا، تكون قد انتهيت!
5. قم بإعادة Repeat ذلك.

إليك طريقة بديلة Alternate.

1. انظر داخل الصندوق.
2. إذا وجدت صندوقًا، انتقل إلى الخطوة 1 مرة أخري.
3. إذا وجدت مفتاحًا، تكون قد انتهيت!

أي طريقة تبدو أسهل بالنسبة لك؟

تستخدم الطريقة الأولي حلقة Loop وهي بينما while.

بينما الكومة ليست فارغة، اسحب صندوقًا وانظر داخله:

```
def look_for_key(main_box):
  pile = main_box.make_a_pile_to_look_through()
  while pile is not empty:
    box = pile.grab_a_box()
    for item in box:
      if item.is_a_box():
        pile.append(item)
      elif item.is_a_key():
        print "found the key!"
```

الطريقة الثانية تستخدم التكرارية Recursion. التكرارية هي التي تَستدعي Call فيها الدالة Function نفسها.

فيما يلي الطريقة الثانية بالكود الزائف Pseudocode:

```
def look_for_key(box):
  for item in box:
    if item.is_a_box():
      look_for_key(item)
    elif item.is_a_key():
      print "found the key!"
```

التكرار Recursion

كلا النهجين Approaches يحققان نفس الشيء، لكن النهج الثاني أوضح بالنسبة لي.

يتم استخدام التكرارية Recursion عندما تجعل الحل Solution أكثر وضوحًا.

لا توجد فائدة بالنسبة للأداء Performance لاستخدام التكرارية Recursion؛ في الواقع، أحيانًا تكون الحلقات Loops أفضل للأداء. يعجبني هذا الاقتباس بواسطة Leigh Caldwell على موقع Stack Overflow:

" الحلقات Loops قد تحقق مكاسب في الأداء Performance لبرنامجك Program. والتكرارية Recursion قد تحقق مكاسب في الأداء Performance للمبرمج Programmer الخاص بك. اختر أيهما أكثر أهمية في حالتك!

العديد من الخوارزميات Algorithms تستخدم التكرارية Recursion، لذلك من المهم فهمها.

الحالة الأساسية Base Case والحالة التكرارية Recursive Case

نظرًا لأن الدالة التكرارية Recursive Function تستدعي Call نفسها، فمن السهل كتابة Write دالة Function بشكل غير صحيح وينتج حلقة لا نهائية Infinite Loop. على سبيل المثال، افترض أنك تريد كتابة دالة Function تطبع Prints عدًا تنازليًا Countdown، مثل هذا:

```
> 3...2...1
```

يمكنك كتابتها بشكل تكراري Recursively، على النحو التالي:

```
def countdown(i):
print i
countdown(i-1)
```

اكتب هذا الكود Code وقم بتشغيله Run. ستلاحظ مشكلة: ستعمل هذه الدالة Function إلى الأبد!

```
> 3...2...1...0...-1...-2...
```

حلقة لا نهائية Infinite Loop

(اضغط على Ctrl-C لإنهاء Kill البرنامج النصي Script)

عندما تكتب دالة تكرارية Recursive Function، عليك أن تخبرها متى تتوقف عن التكرار Recursing. هذا هو السبب في أن كل دالة تكرارية Recursive Function تتكون من جزئين: الحالة الأساسية Base Case والحالة التكرارية Recursive Case. الحالة التكرارية هي عندما تستدعي الدالة نفسها. الحالة الأساسية Base Case هي عندما لا تستدعي الدالة نفسها مرة أخرى ... وبهذا لا تدخل في حلقة لا نهائية Infinite Loop.

دعونا نقوم بإضافة حالة أساسية Base Case إلى دالة Function العد التنازلي Countdown:

```
def countdown(i):
  print i
  if i <= 0:
    return
  else:
    countdown(i-1)
```

الحالة الأساسية Base Case

الحالة التكرارية Recursive Case

الآن تعمل الدالة Function كما هو متوقع. وتكون شيء من هذا القبيل.

الدفتر Stack

هذا القسم يغطي دفتر الاستدعاءات Call Stack. إنه مفهوم Concept مهم في البرمجة Programming. دفتر الاستدعاءات Call Stack يعتبر مفهومًا مهمًا في البرمجة العامة General Programming، ومن المهم أيضًا فهمه عند استخدام التكرارية Recursion.

لنفترض أنك تقيم حفل شواء. تحتفظ بقائمة مهام To-Do List للشواء، على شكل دفتر Stack من الملاحظات اللاصقة Sticky Notes.

تذكر مرة أخرى عندما تحدثنا عن المصفوفات Arrays والقوائم Lists، وكان لديك قائمة مهام To-Do List؟ كان يمكنك إضافة عناصر المهام To-Do Items في أي مكان إلى القائمة List أو حذف Delete عناصر عشوائية Random Items. دفتر Stack الملاحظات اللاصقة أبسط بكثير. عند إدراج Insert عنصر Item، يتم إضافته إلى أعلى القائمة Top. عند قراءة Read عنصر Item، فأنت تقرأ العنصر العلوي فقط Topmost، ويتم سحبه من القائمة. لذا فإن قائمة المهام الخاصة بك تحتوي على إجراءين فقط:

دفع Push (إدراج Insert) وسحب Pop (إزالة Remove وقراءة Read).

دعونا نرى قائمة المهام To-Do أثناء العمل In Action.

هيكل البيانات Data Structure هذا يسمى الدفتر Stack. الدفتر Stack عبارة عن هيكل بيانات بسيط.

لقد كنت تستخدم دفتراً Stack طوال الوقت دون أن تدرك ذلك!

دفتر الاستدعاءات Call Stack

جهاز الكمبيوتر الخاص بك يستخدم دفتراً Stack يسمى داخلياً Internally دفتر الاستدعاءات Call Stack.

دعونا نراه أثناء العمل. فيما يلي دالة بسيطة Simple Function:

```
def greet(name):
  print "hello, " + name + "!"
  greet2(name)
  print "getting ready to say bye..."
  bye()
```

هذه الدالة تقوم بتحيِّتك Greet ثم تستدعي دالتان آخرتان Two Functions. فيما يلي هاتان الدالتان:

```
def greet2(name):
  print "how are you, " + name + "?"

def bye():
  print "ok bye!"
```

دعنا نقوم بجولة فيما يحدث عندما تقوم باستدعاء Call دالة Function.

ملاحظة

`print` هي دالة Function في Python، ولكن لتسهيل الأمور في هذا المثال، نتظاهر بأنها ليست كذلك.

لنفترض أنك استدعيت Call دالة `greet("maggie")`. أولاً، يقوم جهاز الكمبيوتر الخاص بك بتخصيص صندوقاً Box من الذاكرة Memory لهذا الاستدعاء للدالة Function Call.

الآن هيا بنا نستخدم الذاكرة Memory. تم تعيين Set المتغير `name` للقيمة "maggie"، هذا يجب حفظها Saved في الذاكرة Memory.

في كل مرة تقوم فيها باستدعاء دالة Function Call، يحفظ جهاز الكمبيوتر الخاص بك القيم Values لجميع المتغيرات Variables الخاصة بهذا الاستدعاء Call في الذاكرة Memory مثل هذا.

بعد ذلك، تقوم بطباعة Print جملة "hello, maggie!" ثم تقوم باستدعاء Call الدالة greet2("maggie"). مرة أخرى، يقوم جهاز الكمبيوتر الخاص بك بتخصيص صندوق Box من الذاكرة Memory لهذا الاستدعاء للدالة Function Call.

جهاز الكمبيوتر الخاص بك يستخدم دفتراً Stack لهذه الصناديق Boxes. يتم إضافة الصندوق الثاني فوق الصندوق الأول. أنت تقوم بطباعة Print جملة "how are you, maggie?" ثم تعود Return من استدعاء الدالة Function Call. عندما يحدث هذا، يتم سحب Pop الصندوق الموجود أعلى الدفتر Stack.

الآن الصندوق العلوي Topmost Box في الدفتر Stack مخصص لدالة greet، مما يعني أنك عُدت Return إلى دالة greet. عندما استدعيت Call دالة greet2، اكتملت جزئيًا دالة greet. هذه هي الفكرة الكبيرة وراء هذا القسم: عندما تقوم باستدعاء Call دالة Function من دالة Function أخرى، تتوقف مؤقتًا الدالة صاحبة الاستدعاء Calling Function في حالة اكتمال جزئي Partially Completed State. جميع قيم Values المتغيرات Variables الخاصة بهذه الدالة Function لا تزال مخزنة Stored في الذاكرة Memory. الآن بعد أن انتهيت من دالة greet2، ستعود إلى دالة greet، وتستأنف من حيث توقفت. تقوم أولاً بطباعة Print جملة "getting ready to say bye...". تقوم باستدعاء دالة bye.

يتم إضافة صندوق Box لهذه الدالة Function إلى أعلي Top الدفتر Stack. ثم تطبع Print جملة "ok bye!" والعودة Return من استدعاء الدالة Function Call.

وتعود إلى دالة greet. لا يوجد شيء آخر يجب القيام به، لذا أنت تعود Return من دالة greet أيضًا.

هذا الدفتر Stack، المستخدم لحفظ Save المتغيرات Variables لدوال متعددة Multiple Functions، يسمى دفتر الاستدعاءات Call Stack.

التمرين Exercise

3.1 افترض أني أعطيتك دفتر استدعاءات Call Stack مثل هذا.

ما هي المعلومات التي يمكن أن تزودني بها، فقط بناءً على دفتر الاستدعاءات Call Stack هذا؟

الآن دعونا نرى دفتر الاستدعاءات Call Stack أثناء العمل مع دالة تكرارية Recursive Function.

دفتر الاستدعاءات Call Stack مع التكرارية Recursion

الدوال التكرارية Recursive Functions تستخدم أيضا دفتر الاستدعاءات Call Stack! دعونا نلقي نظرة على هذا أثناء العمل من خلال دالة المضروب factorial.

factorial(5) يتم كتابتها بالشكل !5، ويتم حسابه على النحو التالي: 5*4*3*2*1 = !5. بالمثل، مضروب (3) = 3*2*1 = factorial(3).

هذه دالة تكرارية Recursive Function لحساب مضروب رقم Number Factorial:

```
def fact(x):
  if x == 1:
    return 1
  else:
    return x * fact(x-1)
```

الآن أنت تقوم باستدعاء Call الدالة `fact(3)`. دعنا نسير خلال هذا الاستدعاء Call سطرًا بسطر Line By Line ونرى كيف يتغير الدفتر Stack. تذكر أن الصندوق العلوي Topmost Box في الدفتر Stack يخبرنا باستدعاء الدالة `fact` الحالي الذي تعمل عليه.

لاحظ أن كل استدعاء Call للدالة `fact` لها نسختها الخاصة من `x`.

لا يمكنك الوصول Access إلى نسخة `x` لدالة Function مختلفة.

الدفتر Stack يلعب دورًا كبيرًا في التكرارية Recursion.

في المثال الافتتاحي، كان هناك طريقتين للعثور على المفتاح. فيما يلي الطريقة الأولى مرة أخرى.

بهذه الطريقة، يمكنك صنع كومة من الصناديق Boxes للبحث خلالها، بحيث تعرف دائمًا الصناديق التي لا تزال بحاجة إلى البحث داخلها.

ولكن في النهج التكراري Recursive Approach، لا توجد كومة من الصناديق.

إذا لم يكن هناك كومة، كيف تعرف خوارزميتك Algorithm الصناديق Boxes التي لا يزال يتعين عليك البحث داخلها؟ هذا مثال.

في هذه المرحلة، يبدو دفتر الاستدعاءات Call Stack هكذا

يتم حفظ "كومة الصناديق" على الدفتر Stack! هذا دفتر Stack لاستدعاءات الدوال نصف المكتملة Half-Completed Function Calls، ولكل منها قائمة نصف مكتملة Half-Complete List من الصناديق Boxes للبحث خلالها. يعد استخدام الدفتر Stack مناسبًا لأنك لست مضطرًا إلى تتبع Track كومة من الصناديق بنفسك – فالدفتر Stack يقوم بذلك نيابة عنك.

استخدام الدفتر Stack يعد أمرًا ملائمًا، ولكن هناك تكلفة Cost: حفظ كل هذه المعلومات يمكن أن يستهلك قدرًا كبيرًا من الذاكرة Memory. كل واحد من استدعاءات الدوال Function Calls هذه يستهلك بعض الذاكرة، وعندما يكون دفترك Stack طويلًا جدًا، فهذا يعني أن جهاز الكمبيوتر الخاص بك يقوم بحفظ المعلومات للعديد من استدعاءات الدوال Function Calls. في هذه المرحلة، لديك خياران:

- يمكنك إعادة كتابة Re-Write الكود Code الخاص بك لاستخدام حلقة Loop بدلاً من ذلك.
- يمكنك استخدام شيء يسمى التكرار الذّيلي Tail Recursion. هذا موضوع تكراري متقدم Advanced Recursion Topic خارج نطاق هذا الكتاب. وهو أيضًا مدعوم فقط بواسطة بعض لغات البرمجة Programming Languages وليس كلها.

التمرين Exercise

3.2 افترض أنك كتبت بطريق الخطأ دالة تكرارية Recursive Function تعمل إلى الأبد. كما رأيت، يقوم جهاز الكمبيوتر الخاص بك بتخصيص Allocate ذاكرة Memory على الدفتر Stack لكل استدعاء دالة Function Call. ماذا يحدث للدفتر Stack عندما تعمل الدالة التكرارية إلى الأبد؟

الخلاصة

- التكرارية Recursion هي عندما تستدعي Call دالة Function نفسها.
- كل دالة تكرارية Recursive Function لها حالتان:
  الحالة الأساسية Base Case والحالة التكرارية Recursive Case.
- الدفتر Stack له عمليتان Two Operations: الدفع Push والسحب Pop.
- تذهب جميع استدعاءات الدوال Function Calls إلى دفتر الاستدعاءات Call Stack.
- يمكن أن يصبح دفتر الاستدعاءات Call Stack كبيرًا جدًا، مما يستهلك قدرًا كبيرًا من الذاكرة Memory.

4 الترتيب السريع
Quicksort

في هذا الفصل

- تتعلم فَرِّقْ تَسُدْ Divide-And-Conquer. في بعض الأحيان ستواجه مسألة Problem لا يمكن حلها بأي خوارزمية Algorithm تعلمتها. عندما يواجه خبير خوارزمي Algorithmist جيد مسألة مثل هذه، فإنه لا يستسلم. لديه صندوق أدوات Toolbox مليء بالأساليب التي يستخدمها مع المسألة، في محاولة للتوصل إلى حل Solution. فَرِّقْ تَسُدْ Divide-And-Conquer هي أول أسلوب عام تتعلمه.
- تتعرف على الترتيب السريع Quicksort، وهو عبارة عن خوارزمية ترتيب أنيقة Elegant Sorting Algorithm تُستخدم غالبًا في الممارسة In Practice. خوارزمية Quicksort تَستخدم فَرِّقْ تَسُدْ Divide-And-Conquer.

لقد تعلمت التكرارية Recursion في الفصل السابق. يركز هذا الفصل على استخدام مهارتك الجديدة لحل المسائل Solve Problems. سنستكشف فَرِّقْ تَسُدْ (D&C) Divide-And-Conquer، وهي أسلوب تكراري Recursive Technique معروف لحل المسائل Solve Problems.

يدخل هذا الفصل حقًا في صلب الخوارزميات Algorithms. لا تكون الخوارزمية Algorithm مفيدة جدًا إذا تمكنت من حل نوع واحد فقط من المسائل Problems. بدلاً من ذلك، تمنحك فَرِّقْ تَسُدْ (D&C) طريقة جديدة للتفكير في حل المسائل Solve Problems. فَرِّقْ تَسُدْ (D&C) هي أداة أخرى في صندوق الأدوات Toolbox الخاص بك. عندما تحصل على مسألة جديدة New Problem، فلا داعي لأن تكون في حيرة من أمرك. بدلاً من ذلك، يمكنك أن تسأل، "هل يمكنني حلها إذا استخدمت فَرِّقْ تَسُدْ (D&C) Divide-And-Conquer؟"
في نهاية الفصل، ستتعلم أول خوارزمية فَرِّقْ تَسُدْ D&C رئيسية: الترتيب السريع Quicksort. خوارزمية الترتيب السريع Quicksort هي خوارزمية ترتيب Sorting Algorithm، وهي أسرع بكثير من ترتيب التحديد Selection Sort (الذي تعلمته في الفصل الثاني). إنه مثال جيد على الكود الأنيق Elegant Code.

فَرِّقْ تَسُدْ Divide & Conquer

يمكن أن تستغرق فَرِّقْ تَسُدْ (D&C) بعض الوقت لفهمها. لذلك، سنقوم بثلاثة أمثلة. أولاً سأريكم مثالاً مرئياً Visual Example. ثم سأقوم بعمل مثال كود Code Example أقل جمالًا ولكن ربما يكون أسهل. أخيرًا، سنمشي خلال الترتيب السريع Quicksort، وهي خوارزمية ترتيب Sorting Algorithm تستخدم فَرِّقْ تَسُدْ (D&C).

لنفترض أنك مزارع يمتلك قطعة أرض.

تريد تقسيم Divide هذه المزرعة بالتساوي Evenly إلى قطع مربعة Square Plots. تريد أن تكون القطع Plots كبيرة قدر الإمكان. لذلك لن ينجح أي من الحلول التالية.

كيف يمكنك تحديد أكبر حجم مربع Square Size يمكنك استخدامه لقطعة أرض؟ استخدم استراتيجية فَرِّقْ تَسُدْ D&C! خوارزميات فَرِّقْ تَسُدْ D&C هي خوارزميات تكرارية Recursive Algorithms. لحل مسألة Problem باستخدام D&C، هناك خطوتان:

1. اكتشف الحالة الأساسية Base Case. يجب أن تكون أبسط حالة ممكنة Simplest Possible Case.
2. قم بتقسيم Divide مسألتك أو اختزالها Reduce حتى تصبح الحالة الأساسية Base Case.

دعنا نستخدم فَرِّقْ تَسُدْ D&C لإيجاد حل لهذه المسألة. ما هو أكبر حجم مربع يمكنك استخدامه؟

أولاً، اكتشف الحالة الأساسية Base Case. أسهل حالة Easiest Case هي إذا كان أحد الجانبين من مضاعفات Multiple الجانب الآخر.

لنفترض أن أحد الجانبين طوله 25 مترًا والجانب الآخر 50 مترًا. إذاً أكبر مربع يمكنك استخدامه هو 25 م × 25 م. أنت بحاجة إلى اثنين من هذه المربعات لتقسيم الأرض.

الآن أنت بحاجة إلى معرفة الحالة التكرارية Recursive Case. وهنا يأتي دور فَرِّقْ تَسُدْ D&C. وفقًا لـ D&C، مع كل استدعاء تكراري Recursive Call، عليك اختزال Reduce مسألتك Problem. كيف نختزل المسألة هنا؟ لنبدأ بتحديد أكبر المربعات Boxes التي يمكنك استخدامها.

يمكنك احتواء مربعين 640 × 640 هناك، وما زالت هناك بعض الأرض المتبقية تحتاج للتقسيم. هنا تأتي الآن لحظة "آها!". هناك قطعة أرض متبقية تحتاج للتقسيم. لماذا لا نُطبِّق نفس الخوارزمية على هذه القطعة؟

لذلك بدأت بمزرعة 1680 × 640 كنت تريد تقسيمها. لكنك الآن بحاجة إلى تقسيم قطعة أصغر، 640 × 400. إذا وجدت أكبر مربع Biggest Box ينجح مع هذا الحجم، فسيكون هذا هو أكبر مربع ينجح مع الأرض بأكملها. لقد اختزلت Reduced للتو المسألة Problem من أرض 1680 × 640 إلى أرض 640 × 400!

خوارزمية إقليدس Euclid’s Algorithm

"إذا وجدت أكبر مربع ينجح مع هذا الحجم، فسيكون هذا هو أكبر مربع ينجح مع الأرض بأكملها." إذا لم يكن من الواضح لماذا هذه العبارة صحيحة، فلا تقلق. هي بالفعل ليست واضحة. لسوء الحظ، فإن إثبات – برهان Proof سبب نجاحها أطول من أن يتم تضمينه في هذا الكتاب، لذلك عليك فقط أن تصدقني أنه ينجح. إذا كنت تريد فهم الإثبات - البرهان Proof، فابحث عن خوارزمية إقليدس Euclid’s Algorithm.

أكاديمية خان Khan Academy لديها شرح جيد هنا:

https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm

دعونا نقوم بتطبيق نفس الخوارزمية Algorithm مرة أخرى. بدءًا من مزرعة 640 × 400 م، أكبر مربع يمكنك إنشاؤه هو 400 × 400 م.

وهذا يتركك مع قطعة أصغر، 400 × 240 م.

ويمكنك رسم مربع عليها للحصول على قطعة أصغر، 240 × 160 م.

ثم ترسم مربعًا عليها للحصول على قطعة أصغر.

حسناً، أنت وصلت للحالة الأساسية Base Case: 80 هو إحدى عوامل (Factor) لـ 160. إذا قمت بتقسيم هذه القطعة باستخدام مربعات، فلن يتبقى لديك أي شيء!

لذلك، بالنسبة للمزرعة الأصلية، فإن أكبر حجم للقطعة Biggest Plot Size يمكنك استخدامه هو 80 × 80 م.

للتلخيص، فيما يلي كيفية عمل فَرِّقْ تَسُدْ D&C:

1. اكتشف حالة بسيطة Simple Case كحالة أساسية Base Case.
2. اكتشف كيفية اختزال Reduce مسألتك Problem حتى الوصول إلى الحالة الأساسية Base Case.

فَرِّقْ تَسُدْ D&C ليست خوارزمية بسيطة يمكنك تطبيقها على مسألة. بدلاً من ذلك، إنها طريقة للتفكير في مسألة. لنقوم بمثال آخر.

لقد تم إعطاؤك مصفوفة Array من الأرقام Numbers.

يجب عليك جمع كل الأرقام وإرجاع Return الإجمالي Total. من السهل جدًا القيام بذلك باستخدام حلقة Loop:

```
def sum(arr):
  total = 0
  for x in arr:
    total += x
  return total

print sum([1, 2, 3, 4])
```

لكن كيف يمكنك القيام بذلك باستخدام دالة تكرارية Recursive Function؟

الخطوة 1: اكتشف الحالة الأساسية Base Case. ما هي أبسط مصفوفة Simplest Array يمكن أن تحصل عليها؟ فكر في أبسط حالة Simplest Case، ثم تابع القراءة. إذا حصلت على مصفوفة Array تحتوي على 0 أو 1 عنصر Element، فمن السهل جدًا جمعها.

لذلك ستكون هذه هي الحالة الأساسية Base Case.

الخطوة 2: تحتاج إلى تتحرك باتجاه مصفوفة فارغة Empty Array مع كل استدعاء تكراري Recursive Call. كيف تقوم باختزال Reduce حجم مسألتك Problem Size؟ هذه إحدى الطرق.

إنه مثل هذا.

في كلتا الحالتين، تكون النتيجة 12. لكن في الإصدار Version الثاني، تقوم بتمرير Pass مصفوفة Array أصغر إلى دالة الجمع sum. أي أنك اختزلت من حجم مسألتك!

يمكن أن تعمل دالة sum على النحو التالي.

ها هي أثناء العمل In Action.

تذكّر، التكرارية Recursion تقوم بتتبع Track الحالة State.

نصيحة

عندما تكتب دالة تكرارية Recursive Function تتضمن مصفوفة Array، فإن الحالة الأساسية Base Case غالبًا ما تكون مصفوفة فارغة Empty Array أو مصفوفة بها عنصر واحد. إذا واجهتك مشكلة، فجرب ذلك أولاً.

نظرة خاطفة على البرمجة الدالِّية (باستخدام الدوال Functions)
Functional Programming

"لماذا أفعل هذا بشكل تكراري Recursively إذا كان بإمكاني القيام بذلك بسهولة باستخدام حلقة Loop؟" قد تفكر في ذلك. حسنًا، هذه نظرة خاطفة على البرمجة الدالِّية Functional Programming! لغات البرمجة الدالِّية مثل Haskell ليس لديها حلقات Loops، لذلك عليك استخدام التكرار Recursion لكتابة دوال Functions مثل هذه. إذا كان لديك فهم جيد للتكرارية Recursion، فسيكون تعلم اللغات الدالِّية Functional Programming أسهل. على سبيل المثال، فيما يلي كيفية كتابة دالة جمع sum في لغة Haskell:

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

الحالة الأساسية Base Case

الحالة التكرارية Recursive Case

لاحظ أنه يبدو أن لديك تعريفين Two Definitions للدالة Function. يتم تشغيل التعريف الأول عند الوصول إلى الحالة الأساسية Base Case. التعريف الثاني يتم تشغيله في الحالة التكرارية Recursive Case. يمكنك أيضًا كتابة هذه الدالة في لغة Haskell باستخدام جملة If Statement:

```
sum arr = if arr == []
            then 0
            else (head arr) + (sum (tail arr))
```

لكن التعريف الأول أسهل في القراءة. نظرًا لأن لغة Haskell تستخدم التكرارية Recursion بشكل مكثف، فإنها تتضمن جميع أنواع التفاصيل الدقيقة مثل هذه لتسهيل عملية التكرار Recursion. إذا كنت تحب التكرارية Recursion، أو كنت مهتمًا بتعلم لغة جديدة، قم بالاطلاع على لغة Haskell.

التمارين Exercises

4.1 اكتب كود Code لدالة sum السابقة.

4.2 اكتب دالة تكرارية Recursive Function لحساب عدد العناصر Items في قائمة List.

4.3 اعثر علي أكبر عدد Maximum Number في قائمة List.

4.4 تذكّر البحث الثنائي Binary Search من الفصل الأول؟ إنها خوارزمية فَرِّقْ تَسُدْ Divide-And-Conquer Algorithm أيضًا. هل يمكنك الوصول إلى الحالة الأساسية Base Case والحالة التكرارية Recursive Case للبحث الثنائي Binary Search؟

الترتيب السريع Quicksort

الترتيب السريع Quicksort هو خوارزمية ترتيب Sorting Algorithm. إنها أسرع بكثير من ترتيب التحديد Selection Sort ويتم استخدامها بشكل متكرر في الحياة الواقعية. على سبيل المثال، مكتبة لغة C القياسية (C Standard Library) لديها دالة Function تسمى `qsort`، وهي التنفيذ Implementation الخاص بها للترتيب السريع Quicksort. الترتيب السريع Quicksort يستخدم أيضًا فَرِّقْ تَسُدْ (D&C).

دعونا نستخدم الترتيب السريع Quicksort لترتيب Sort مصفوفة Array. ما هي أبسط مصفوفة يمكن لخوارزمية ترتيب Sorting Algorithm التعامل معها (تذكر نصيحتي من القسم السابق)؟ حسنًا، بعض المصفوفات Arrays لا تحتاج أن يتم ترتيبها Sorted على الإطلاق.

المصفوفات الفارغة Empty Arrays والمصفوفات التي تحتوي على عنصر واحد فقط ستكون هي الحالة الأساسية Base Case. يمكنك فحسب إرجاع Return هذه المصفوفات كما لو أنه لا يوجد شيء لترتيبه Sort:

```
def quicksort(array):
  if len(array) < 2:
    return array
```

دعونا نلقي نظرة على مصفوفات Arrays أكبر. من السهل جدًا أيضًا ترتيب مصفوفة Array مكونة من عنصرين.

ماذا عن مصفوفة Array مكونة من ثلاث عناصر Elements؟

تذكر أنك تستخدم فَرِّقْ تَسُدْ (Divide-And-Conquer (D&C. لذلك تريد تقسيم هذه المصفوفة Array حتى تصل إلى الحالة الأساسية Base Case. إليك كيفية عمل الترتيب السريع Quicksort.

أولاً، اختر عنصرًا من المصفوفة. هذا العنصر Element يسمى المحور Pivot.

سنتحدث عن كيفية اختيار محور Pivot جيد لاحقًا. في الوقت الحالي، لنقول أن العنصر Item الأول في المصفوفة Array هو المحور Pivot.

اعثر الآن على العناصر الأصغر من المحور Pivot والعناصر الأكبر من المحور Pivot.

وهذا ما يسمى بالتقسيم Partitioning. الآن لديك

- مصفوفة فرعية Sub-Array من جميع الأعداد Numbers الأصغر من المحور Pivot
- المحور Pivot
- مصفوفة فرعية Sub-Array من جميع الأعداد الأكبر من المحور Pivot

لم يتم ترتيب Sort المصفوفتين الفرعيتين Two Sub-Arrays. لقد تم تقسيمهم Partitioned فقط. ولكن إذا تم ترتيبهم Sorted، فسيكون من السهل جدًا ترتيب Sorting المصفوفة بأكملها Whole Array.

إذا تم ترتيب Sort المصفوفات الفرعية Sub-Arrays، إذن يمكنك جمع Combine كل شيء على هذا النحو - `left array + pivot + right array` - وستحصل على مصفوفة مرتبة Sorted Array.

في هذه الحالة، تكون `[10, 15, 33] = [] + [33] + [10, 15]`، وهي مصفوفة مرتبة.

كيف تقوم بترتيب Sort المصفوفات الفرعية Sub-Arrays؟ حسنًا، الحالة الأساسية Base Case للترتيب السريع Quicksort تعرف بالفعل كيفية ترتيب المصفوفات Sort Arrays المكونة من عنصرين (المصفوفة الفرعية اليسرى Left Sub-Array) والمصفوفات الفارغة Empty Arrays (المصفوفة الفرعية اليمنى Right Sub-Array). لذا، إذا قمت باستدعاء Call الترتيب السريع Quicksort على المصفوفتين الفرعيتين Two Sub-Arrays ثم قمت بجمع Combine النتائج، فستحصل على مصفوفة مرتبة Sorted Array!

```
quicksort([15, 10]) + [33] + quicksort([])
> [10, 15, 33]
```

مصفوفة مرتبة Sorted Array

سينجح هذا مع أي محور Pivot. افترض أنك اخترت رقم 15 كمحور Pivot بدلاً من ذلك.

كلتا المصفوفتين الفرعيتين Both Sub-Arrays لديها عنصر واحد فقط، وأنت تعرف كيف تقوم بترتيبها Sort.

لذلك أنت تعرف الآن كيفية ترتيب مصفوفة من ثلاثة عناصر. فيما يلي الخطوات:

1. اختر محورًا Pivot.
2. قم بتقسيم Partition المصفوفة Array إلى مصفوفتين فرعيتين Two Sub-Arrays: عناصر أصغر Less من المحور Pivot وعناصر أكبر Greater من المحور Pivot.
3. قم باستدعاء الترتيب السريع Quicksort بشكل تكراري Recursively على المصفوفتين الفرعيتين Two Sub-Arrays.

ماذا عن مصفوفة Array من أربعة عناصر Elements؟

لنفترض أنك اخترت الرقم 33 كمحور Pivot مرة أخرى.

المصفوفة Array الموجودة على اليسار Left لديها ثلاثة عناصر Elements. أنت تعرف بالفعل كيفية ترتيب مصفوفة من ثلاثة عناصر: قم باستدعاء الترتيب السريع Quicksort عليها بشكل تكراري Recursively.

لذا يمكنك ترتيب Sort مصفوفة Array من أربعة عناصر. وإذا كان بإمكانك ترتيب مصفوفة من أربعة عناصر، فيمكنك ترتيب مصفوفة من خمسة عناصر. لماذا هذا؟

افترض أن لديك هذه المصفوفة من خمسة عناصر.

فيما يلي جميع الطرق التي يمكنك من خلالها تقسيم Partition هذه المصفوفة Array، اعتمادًا على المحور Pivot الذي تختاره.

لاحظ أن كل هذه المصفوفات الفرعية Sub-Arrays لديها ما بين 0 و 4 عناصر. وأنت تعرف بالفعل كيفية ترتيب Sort مصفوفة Array من 0 إلى 4 عناصر باستخدام الترتيب السريع Quicksort! لذلك بغض النظر عن المحور Pivot الذي تختاره، يمكنك استدعاء Call الترتيب السريع Quicksort بشكل تكراري Recursively على المصفوفتين الفرعيتين Two Sub-Arrays.

على سبيل المثال، لنفترض أنك قمت باختيار الرقم 3 كمحور Pivot. يمكنك استدعاء Call الترتيب السريع Quicksort على المصفوفات الفرعية Sub-Arrays.

يتم ترتيب Sort المصفوفات الفرعية Sub-Arrays، ثم تقوم بجمع Combine كل شيء للحصول على مصفوفة مرتبة Sorted Array. ينجح هذا حتى إذا اخترت الرقم 5 كمحور Pivot.

هذا ينجح مع أي عنصر Element كمحور Pivot. لذا يمكنك ترتيب مصفوفة من خمسة عناصر. باستخدام نفس المنطق Logic، يمكنك ترتيب مصفوفة من ستة عناصر، وهكذا.

البراهين الاستقرائية Inductive Proofs

لقد قمت للتو بنظرة خاطفة إلى البراهين (الإثباتات) الاستقرائية Inductive Proofs! البراهين الاستقرائية هي إحدى الطرق لإثبات أن خوارزميتك Algorithm تنجح. كل برهان استقرائي Inductive Proof لديه خطوتين: الحالة الأساسية Base Case والحالة الاستقرائية Inductive Case. تبدو مألوفة؟ على سبيل المثال، افترض أنني أريد إثبات أنه يمكنني الصعود إلى قمة سلم. في الحالة الاستقرائية Inductive Case، إذا كانت قدمي على درجة سلم، يمكنني وضع قدمي على الدرجة التالية. لذا إذا كنت أقف على الدرجة الثانية، فيمكنني الصعود للدرجة الثالثة. هذه هي الحالة الاستقرائية Inductive Case. بالنسبة للحالة الأساسية Base Case، سأقول إن قدمي على الدرجة الأولي. ولذلك، يمكنني صعود السلم بأكمله، الصعود درجة واحدة في كل مرة. أنت تستخدم نفس المنطق Reasoning للترتيب السريع Quicksort. في الحالة الأساسية Base Case، أوضحت أن الخوارزمية تنجح مع الحالة الأساسية: مصفوفات Arrays بالحجم 0 و 1. في الحالة الاستقرائية Inductive Case، أوضحت أنه إذا كان الترتيب السريع Quicksort ينجح مع مصفوفة Array من الحجم 1، فسينجح مع مصفوفة من الحجم 2. وإذا كانت تنجح مع مصفوفة من الحجم 2، فستنجح مع مصفوفات من الحجم 3، وهكذا. ثم يمكنني القول إن الترتيب السريع Quicksort سينجح مع جميع المصفوفات من أي حجم. لن أتعمق أكثر في البراهين الاستقرائية Inductive Proofs هنا، لكنها ممتعة وتسير جنبًا إلى جنب مع فرِّق تَسُد (D&C) Divide & Conquer.

فيما يلي كود Code الترتيب السريع Quicksort:

```
def quicksort(array):
  if len(array) < 2:
    return array
  else:
    pivot = array[0]
    less = [i for i in array[1:] if i <= pivot]

    greater = [i for i in array[1:] if i > pivot]

    return quicksort(less) + [pivot] + quicksort(greater)

print quicksort([10, 5, 2, 3]
```

الحالة الأساسية Base Case: المصفوفات التي تحتوي على 0 أو 1 من العناصر مرتبة Sorted بالفعل

الحالة التكرارية Recursive Case

مصفوفة فرعية Sub-Array من جميع العناصر الأكبر من المحور Pivot

مصفوفة فرعية Sub-Array من جميع العناصر الأصغر من المحور Pivot

خوارزمية الترتيب السريع Quicksort فريدة Unique لأن سرعتها تعتمد على المحور Pivot الذي تختاره. قبل أن أتحدث عن الترتيب السريع، دعونا نلقي نظرة مرة أخرى على أوقات تشغيل Big O الأكثر شيوعًا.

التقديرات Estimates قائمة علي جهاز كمبيوتر بطيء ينفذ 10 عمليات Operations في الثانية

أوقات الأمثلة في هذا المخطط Chart هي تقديرات Estimates إذا أجريت 10 عمليات Operations في الثانية. هذه الرسوم البيانية Graphs ليست دقيقة - فهي موجودة فحسب لتعطيك فكرة عن مدى اختلاف أوقات التشغيل Run Times هذه. في الواقع، يمكن لجهاز الكمبيوتر الخاص بك إجراء أكثر من 10 عمليات في الثانية.

كل وقت تشغيل لديه أيضًا خوارزمية مرفقة Attached Algorithm كمثال Example. تحقق من ترتيب التحديد Selection Sort، الذي تعلمته في الفصل الثاني. وقت التشغيل الخاص به $O(n^2)$. هذه خوارزمية بطيئة جدًا.

هناك خوارزمية ترتيب Sorting Algorithm أخرى تسمى ترتيب الدمج Merge Sort، وقت التشغيل الخاص بها $O(n \log n)$. أسرع بكثير! الترتيب السريع Quicksort حالة محيِّرة – مخادعة Tricky. في أسوأ حالة Worst Case، يستغرق الترتيب السريع Quicksort وقت $O(n^2)$.

إنها بطيئة Slow مثل ترتيب التحديد Selection Sort! لكن هذه أسوأ حالة Worst Case. في الحالة المتوسطة Average Case، يستغرق الترتيب السريع Quicksort وقت $O(n \log n)$. لذلك قد تتساءل:

- ماذا تعني أسوأ حالة Worst Case والحالة المتوسطة Average Case هنا؟
- إذا كان وقت تشغيل الترتيب السريع Quicksort هو $O(n \log n)$ في المتوسط Average، ولكن ترتيب الدمج Merge Sort هو $O(n \log n)$ دائمًا ، فلماذا لا تستخدم ترتيب الدمج Merge Sort؟ أليس أسرع؟

ترتيب الدمج Merge Sort في مقابل الترتيب السريع Quicksort

افترض أن لديك هذه الدالة البسيطة Simple Function لطباعة Print كل عنصر Item في قائمة List:

```
def print_items(list):
  for item in list:
    print item
```

هذه الدالة Function تقوم بالمرور خلال كل عنصر Item في القائمة List وتقوم بطباعته Print. نظرًا لأنها تقوم بالمرور Loop على القائمة بأكملها مرة واحدة، تستغرق هذه الدالة وقت O(n). الآن، افترض أنك قمت بتغيير هذه الدالة Function بحيث تنام Sleep لمدة ثانية واحدة قبل أن تطبع عنصرًا:

```
from time import sleep
def print_items2(list):
  for item in list:
    sleep(1)
    print item
```

قبل أن تطبع عنصرًا، ستتوقف مؤقتًا Pause لمدة ثانية واحدة. افترض أنك قمت بطباعة قائمة من خمسة عناصر باستخدام كلتا الدالتين Both Functions.

كلتا الدالتين Both Functions تقوم بالمرور Loop على القائمة مرة واحدة، لذا فهما يستغرقان وقت O(n). أيهما تعتقد أنه سيكون أسرع في الممارسة In Practice؟ أعتقد أن `print_items` ستكون أسرع بكثير لأنها لا تتوقف مؤقتًا Pause لمدة ثانية واحدة قبل طباعة أي عنصر. لذلك على الرغم من أن كلتا الدالتين لهما نفس السرعة Speed في تدوين Big O Notation، إلا أن `print_items` أسرع في الممارسة العملية.

عندما تكتب تدوين Big O مثل O(n)، فهو يعني ذلك حقًا.

c هو مقدار ثابت من الوقت تستغرقه الخوارزمية Algorithm. إنه يسمى الثابت Constant.

على سبيل المثال، قد يكون `10 milliseconds * n` للدالة `print_items` في مقابل `1 second * n` للدالة `print_items2`.

عادة ما تتجاهل هذا الثابت Constant، لأنه إذا كانت خوارزميتان لهما أوقات Big O مختلفة، فإن الثابت لا يهم حقاً. خذ البحث الثنائي Binary Search والبحث البسيط Simple Search، على سبيل المثال. افترض أن كلا الخوارزميتين لهما هذه الثوابت Constants.

10ms * n	1sec * log n
SIMPLE SEARCH	BINARY SEARCH

قد تقول، "واو Wow! البحث البسيط Simple Search لديه ثابت Constant مقداره 10 مللي ثانية Milliseconds، لكن البحث الثنائي Binary Search لديه ثابت Constant مقداره هو 1 ثانية. البحث البسيط Simple Search أسرع بكثير! " لنفترض الآن أنك تبحث قائمة List من 4 بلايين عنصر. فيما يلي الأوقات.

SIMPLE SEARCH	10ms * 4 BILLION	=	463 days
BINARY SEARCH	1sec * 32	=	32 seconds

كما ترى، لا يزال البحث الثنائي Binary Search أسرع بكثير. هذا الثابت Constant لم يحدث فرقًا على الإطلاق.

لكن في بعض الأحيان يمكن للثابت Constant أن يُحدث فرقًا. الترتيب السريع Quicksort في مقابل ترتيب الدمج Merge Sort أحد الأمثلة. الترتيب السريع Quicksort له ثابت Constant أصغر من ترتيب الدمج Merge Sort. لذلك إذا كانا وقت تشغيل كليهما $O(n \log n)$، يكون الترتيب السريع Quicksort أسرع.

الترتيب السريع Quicksort أسرع من الناحية العملية لأنه يصل إلى الحالة المتوسطة Average Case غالباً أكثر من أسوأ حالة Worst Case.

إذن أنت الآن تتساءل: ما هي الحالة المتوسطة Average Case في مقابل أسوأ حالة Worst Case؟

الحالة المتوسطة Average Case في مقابل أسوأ حالة Worst Case

أداء Performance الترتيب السريع Quicksort يعتمد بشكل كبير على المحور Pivot الذي تختاره. لنفترض أنك تختار دائمًا العنصر الأول كمحور Pivot. وتقوم باستدعاء Call الترتيب السريع Quicksort مع مصفوفة Array تم ترتيبها Sorted بالفعل. الترتيب السريع Quicksort لا يقوم بالتحقق Check لمعرفة ما إذا كانت مصفوفة المدخلات Input Array قد تم ترتيبها بالفعل. لذلك ستظل تحاول ترتيبها Sort.

لاحظ كيف لا تقوم بتقسيم المصفوفة إلى نصفين. بدلاً من ذلك، تكون إحدى المصفوفات الفرعية Sub-Arrays فارغة Empty دائمًا. لذا فإن دفتر الاستدعاءات Call Stack طويل حقًا. الآن بدلاً من ذلك، افترض أنك اخترت دائمًا العنصر الأوسط Middle Element كمحور Pivot. انظر إلى دفتر الاستدعاءات Call Stack الآن.

إنه قصير جدا! نظرًا لأنك تقوم بتقسيم Divide المصفوفة Array إلى النصف في كل مرة، أنت لا تحتاج إلى إجراء العديد من الاستدعاءات التكرارية Recursive Calls. أنت تصل إلى الحالة الأساسية Base Case في وقت أقرب، ويكون دفتر الاستدعاءات Call Stack أقصر بكثير.

المثال الأول الذي رأيته هو سيناريو أسوأ حالة Worst-Case Scenario، والمثال الثاني هو سيناريو أفضل حالة Best-Case Scenario. في أسوأ حالة Worst Case، يكون حجم الدفتر Stack Size هو $O(n)$. في أفضل حالة Best-Case، يكون حجم الدفتر Stack Size هو $O(\log n)$.

الآن انظر إلى المستوى الأول First Level في الدفتر Stack. تختار عنصرًا واحدًا كمحور Pivot، ويتم تقسيم العناصر المتبقية إلى مصفوفات فرعية Sub-Arrays. أنت تقوم بالمرور على كل العناصر الثمانية في المصفوفة Array. لذا فإن العملية الأولى هذه تستغرق وقت $O(n)$. لقد قمت بالمرور على جميع العناصر الثمانية في هذا المستوى Level من دفتر الاستدعاءات Call Stack. لكن في الواقع، أنت تقوم بالمرور على عناصر عددها $O(n)$ في كل مستوى Level من دفتر الاستدعاءات Call Stack.

حتى إذا قمت بتقسيم Partition المصفوفة Array بشكل مختلف، فإنك لا تزال تقوم بالمرور على عناصر O(n) في كل مرة.

لذلك يستغرق كل مستوى Level وقت $O(n)$ لإكماله.

في هذا المثال، توجد مستويات عددها $O(\log n)$ (الطريقة التقنية - الفنية Technical Way لقول ذلك، "أن ارتفاع Height دفتر الاستدعاءات Call Stack هو $O(\log n)$"). ويستغرق كل مستوى Level وقت $O(n)$.

ستستغرق الخوارزمية Algorithm بأكملها وقت $O(n \log n) = O(\log n) * O(n)$.

هذا هو سيناريو أفضل حالة Best-Case Scenario.

في أسوأ حالة Worst Case، هناك مستويات عددها $O(n)$، لذلك ستستغرق الخوارزمية وقت $O(n^2) = O(n) * O(n)$.

حسنا خمن ماذا؟ أنا هنا لأخبرك أن أفضل حالة Best Case هي أيضًا الحالة المتوسطة Average Case. إذا اخترت دائمًا عنصرًا عشوائيًا Random Element في المصفوفة Array كمحور Pivot، فإن الترتيب السريع Quicksort سيكتمل Complete في وقت $O(n \log n)$ في المتوسط Average. الترتيب السريع Quicksort هي واحدة من أسرع خوارزميات الترتيب Sorting Algorithms المتوفرة، وهي مثال جيد جدًا على فرِّق تَسُد Divide & Conquer (D&C).

التمارين Exercises

كم من الوقت ستستغرق كل من هذه العمليات Operations في تدوين Big O Notation؟

4.5 طباعة Printing قيمة Value كل عنصر Element في مصفوفة Array.

4.6 مضاعفة Doubling قيمة Value كل عنصر في مصفوفة.

4.7 مضاعفة قيمة العنصر الأول فقط في مصفوفة.

4.8 إنشاء جدول الضرب Multiplication Table بكل العناصر في المصفوفة. لذا إذا كانت المصفوفة الخاصة بك هي [2, 3, 7, 8, 10]، عليك أولاً ضرب Multiply كل عنصر في الرقم 2، ثم ضرب كل عنصر في الرقم 3، ثم في الرقم 7، وهكذا.

الخلاصة

- فرِّق تَسُد (D&C) تعمل عن طريق تقسيم المسألة Problem إلى أجزاء أصغر وأصغر. إذا كنت تستخدم فرِّق تَسُد D&C على قائمة List، فمن المحتمل أن تكون الحالة الأساسية Base Case عبارة عن مصفوفة فارغة Empty Array أو مصفوفة تحتوي على عنصر واحد.
- إذا كنت تقوم بتنفيذ Implement الترتيب السريع Quicksort، فاختر عنصرًا عشوائيًا Random Element ليكون المحور Pivot. متوسط Average وقت تشغيل Runtime الترتيب السريع Quicksort هو O(n log n)!
- يمكن أن يكون للثابت Constant في تدوين Big O Notation أهمية في بعض الأحيان. هذا هو السبب في أن الترتيب السريع Quicksort أسرع من ترتيب الدمج Merge Sort.
- تقريبا لا يهم أبدًا الثابت Constant في البحث البسيط Simple Search في مقابل البحث الثنائي Binary Search، لأن O(log n) أسرع بكثير من O(n) عندما تصبح قائمتك List كبيرة.

Coding عربي

5 جداول التجزئة
Hash Tables

في هذا الفصل

- ستتعرف على جداول التجزئة (هاش) Hash Tables، واحدة من أكثر هياكل البيانات الأساسية Basic Data Structures المفيدة. جداول التجزئة (هاش) Hash Tables لها استخدامات عديدة؛ يغطي هذا الفصل حالات الاستخدام Use Cases الشائعة.
- تتعرف على التفاصيل الداخلية Internals لجداول التجزئة (هاش) Hash Tables: التنفيذ Implementation والتصادمات Collisions ودوال التجزئة Hash Functions. سيساعدك هذا على فهم كيفية تحليل Analyze أداء Performance جدول التجزئة (هاش) Hash Table.

افترض أنك تعمل في محل بقالة. عندما يشتري العميل منتجًا، عليك البحث عن السعر في كتاب. إذا كان الكتاب غير مرتب أبجديًا Unalphabetized، فقد يستغرق منك الأمر وقتًا طويلاً للبحث في كل سطر عن Apple. ستجري بحثًا بسيطًا Simple Search من الفصل الأول، حيث يتعين عليك إلقاء نظرة على كل سطر Line. هل تتذكر كم من الوقت سيستغرق ذلك؟ $O(n)$. إذا كان الكتاب مرتبًا أبجديًا Alphabetized، فيمكنك إجراء بحث ثنائي Binary Search للعثور على سعر تفاحة Apple. لن يستغرق ذلك سوى وقت $O(\log n)$.

للتذكير، هناك فرق كبير بين وقت O(n) و O(log n)! افترض أنه يمكنك البحث خلال 10 أسطر من الكتاب في الثانية. فيما يلي المدة التي يستغرقها البحث البسيط Simple Search والبحث الثنائي Binary Search.

عدد العناصر في الكتاب — وقت التشغيل

# OF ITEMS IN THE BOOK	O(n)	O(log n)	
100	10 sec	1 sec	تحتاج للتحقق ← $Log_2 100 = 7$ LINES
1000	1.66 min	1 sec	← NEED TO CHECK $Log_2 1000 = 10$ LINES
10000	16.6 min	2 sec	← $Log_2 10000 = 14$ LINES $= 2$ sec

أنت تعلم بالفعل أن البحث الثنائي Binary Search سريع جدا. ولكن بصفتك أمين الصندوق Cashier، فإن البحث عن الأشياء في كتاب يكون مؤلم، حتى لو تم ترتيب Sorted الكتاب. يمكنك أن تشعر بالعميل وهو يغلي بينما تبحث عن عناصر Items في الكتاب. ما تحتاجه حقًا هو صديق لديه جميع الأسماء والأسعار محفوظة في ذاكرته. وبعد ذلك لا تحتاج إلى البحث عن أي شيء: تسأله فحسب، ويخبرك بالإجابة على الفور.

يمكن أن تعطيك صديقتك Maggie السعر في وقت $O(1)$ لأي عنصر، بغض النظر عن حجم الكتاب. إنها أسرع من البحث الثنائي Binary Search.

عدد العناصر في الكتاب # OF ITEMS IN THE BOOK	SIMPLE SEARCH O(n)	BINARY SEARCH O(Log n)	MAGGIE O(1)
100	10 sec	1 sec	INSTANT
1000	1.6 min	1 sec	INSTANT
10000	16.6 min	2 sec	INSTANT

يا لها من شخص رائع! كيف تحصل على صديق مثلها؟

دعونا نرتدي قبعات هياكل البيانات Data Structures. أنت تعرف هيكلا بيانات Two Data Structures حتى الآن: المصفوفات Arrays والقوائم Lists (لن أتحدث عن الدفاتر Stacks لأنك لا تستطيع حقًا "البحث Search" عن شيء ما في الدفتر Stack). يمكنك تنفيذ Implement هذا الكتاب كمصفوفة Array.

(EGGS, 2.49)	(MILK, 1.49)	(PEAR, 0.79)

كل عنصر Item في المصفوفة Array هو في الواقع عنصرين Two Items: أحدهما هو اسم Name نوع من المنتجات، والآخر هو السعر Price. إذا قمت بترتيب Sort هذه المصفوفة Array من حيث By Name، يمكنك إجراء بحث ثنائي Binary Search عليها للعثور على سعر Price العنصر Item. إذاً يمكنك العثور على العناصر Items في وقت $O(\log n)$. لكنك تريد العثور على العناصر في وقت $O(1)$. هذا يعني أنك تريد أن تصنع نسخة من Maggie. وهنا يأتي دور دوال التجزئة (هاش) Hash Functions.

دوال التجزئة Hash Functions

دالة التجزئة (هاش) Hash Function هي دالة تضع فيها سلسلة حروف String ثم تقوم بإرجاع Return رقم Number.

في المصطلحات التقنية - الفنية Technical Terminology، يمكننا القول إن دالة التجزئة (هاش) Hash Function "تقوم بتعيين Mapping سلاسل الحروف Strings إلى أرقام Numbers". قد تعتقد أنه لا يوجد نمط Pattern يمكن تمييزه للرقم Number الذي تحصل عليه عندما تضع سلسلة حروف String. ولكن هناك بعض المتطلبات Requirements لدالة التجزئة (هاش) Hash Function:

- يجب على دالة التجزئة أن تكون متسقة Consistent. على سبيل المثال، لنفترض أنك وضعت كلمة "apple" وحصلت على رقم "4". في كل مرة تضع فيها كلمة "apple"، يجب أن تحصل على رقم "4" مرة أخرى. بدون هذا، لن ينجح جدول التجزئة (هاش) Hash Table الخاص بك.
- يجب على دالة التجزئة أن تقوم بتعيين Map الكلمات المختلفة Different Words لأرقام مختلفة Different Numbers. على سبيل المثال، لا تكون دالة التجزئة (هاش) Hash Function جيدة إذا كانت تُرجع دائمًا رقم "1" لأي كلمة Word تضعها. في أفضل حالة Best Case، يجب تعيين Map كل كلمة Word مختلفة إلى رقم Number مختلف.

إذاً تقوم دالة التجزئة (هاش) Hash Function بتعيين Map سلاسل الحروف Strings إلى أرقام Numbers.

ما فائدة ذلك؟ حسنًا، يمكنك استخدامها لصنع Maggie خاصة بك.

ابدأ بمصفوفة فارغة Empty Array:

ستقوم بتخزين Store جميع أسعارك Prices في هذه المصفوفة Array. هيا نقوم بإضافة Add سعر تفاحة. قم بتغذية Feed كلمة "apple" في دالة التجزئة (هاش) Hash Function.

تقوم دالة التجزئة (هاش) Hash Function بإخراج Output الرقم "3". لذلك دعونا نقوم بتخزين سعر تفاحة في الفهرس Index رقم 3 في المصفوفة Array.

هيا نقوم بإضافة الحليب. قم بتغذية Feed كلمة "milk" في دالة التجزئة (هاش) Hash Function.

تقول دالة التجزئة (هاش) Hash Function الرقم "0". دعنا نخزن سعر الحليب في الفهرس Index رقم 0.

استمر، وفي النهاية ستكون المصفوفة Array بأكملها مليئة بالأسعار Prices.

الآن تسأل، "حسناً، ما هو سعر ثمرة الأفوكادو avocado؟" لا تحتاج للبحث عنه في المصفوفة Array. قم فحسب بتغذية Feed كلمة "avocado" في دالة التجزئة (هاش) Hash Function.

سوف تخبرك دالة التجزئة أن السعر مُخزَّن Stored في الفهرس Index رقم 4. ومن المؤكد أنه هناك.

تخبرك دالة التجزئة (هاش) Hash Function بمكان تخزين السعر بالضبط Exactly، لذلك لا داعي للبحث على الإطلاق! هذا ينجح لأن

- دالة التجزئة Hash Function تقوم بشكل متسق Consistently بتعيين Map الاسم Name إلى نفس الفهرس Index. في كل مرة تضع فيها كلمة "avocado"، ستحصل على نفس الرقم Number مرة أخرى. لذا يمكنك استخدامه في المرة الأولى للعثور على مكان لتخزين سعر الأفوكادو avocado، وبعد ذلك يمكنك استخدامه للعثور على المكان الذي قمت بتخزين هذا السعر فيه.
- تقوم دالة التجزئة Hash Function بتعيين Map سلاسل حروف مختلفة Different Strings لفهارس مختلفة Different Indexes. يتم تعيين كلمة "avocado" للفهرس رقم 4. ويتم تعيين كلمة "milk" للفهرس رقم 0. كل شيء يتم تعيينه Mapped إلى خانة مختلفة Different Slot في المصفوفة Array حيث يمكنك تخزين Store سعره.
- تعرف دالة التجزئة Hash Function حجم مصفوفتك Array وتقوم فقط بإرجاع Return الفهارس الصالحة Valid Indexes. لذلك إذا كانت المصفوفة الخاصة بك مكونة من 5 عناصر، فإن دالة التجزئة لا تقوم بإرجاع 100 ... لن يكون هذا فهرسًا صالحًا Valid Index في المصفوفة.

لقد قمت للتو بصنع Maggie! ضع دالة تجزئة Hash Function ومصفوفة Array معًا، وستحصل على هيكل بيانات Data Structure يسمى جدول التجزئة (هاش) Hash Table. جدول التجزئة هو هيكل البيانات الأول الذي سوف تتعلمه الذي لديه بعض المنطق Logic الإضافي وراءه. تقوم المصفوفات Arrays والقوائم Lists بالتعيين Map مباشرة إلى الذاكرة Memory، لكن جداول التجزئة Hash Tables أكثر ذكاءً. فهي تستخدم دالة تجزئة Hash Function لمعرفة مكان تخزين العناصر بذكاء.

من المحتمل أن تكون جداول التجزئة Hash Tables هي هيكل البيانات المعقد Complex Data Structure الأكثر فائدة التي ستتعلمها. وهي تُعرف أيضًا باسم تعيينات التجزئة Hash Maps والتعيينات Maps والقواميس Dictionaries والمصفوفات الترابطية Associative Arrays. وجداول التجزئة سريعة! تذكر مناقشتنا للمصفوفات Arrays والقوائم المرتبطة Linked Lists مرة أخرى في الفصل الثاني؟ يمكنك الحصول على عنصر Item من مصفوفة Array على الفور Instantly. وتستخدم جداول التجزئة Hash Tables مصفوفة Array لتخزين Store البيانات Data، ولذلك فهما يستويان في السرعة (جداول التجزئة والمصفوفات).

ربما لن تضطر مطلقًا إلى تنفيذ Implement جداول التجزئة Hash Tables بنفسك. أي لغة جيدة سيكون لها تنفيذ - تطبيق Implementation لجداول التجزئة. تحتوي بايثون Python على جداول تجزئة Hash Tables؛ يُطلق عليها قواميس Dictionaries.

يمكنك عمل جدول تجزئة جديد New Hash Table باستخدام دالة `dict`:

```
>>> book = dict()
```

`book` هو جدول تجزئة جديد New Hash Table. دعنا نضيف بعض الأسعار لـ `book`:

```
>>> book["apple"] = 0.67
>>> book["milk"] = 1.49
>>> book["avocado"] = 1.49
>>> print book
{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}
```

سعر التفاحة 67 سنتا

الحليب يكلف 1.49 دولار

سهل جدا! الآن دعنا نسأل عن سعر ثمرة الأفوكادو avocado:

```
>>> print book["avocado"]
1.49
```

سعر ثمرة الأفوكادو

جدول التجزئة Hash Table لديه مفاتيح Keys وقيم Values. في تجزئة Book، تكون أسماء المنتجات هي المفاتيح Keys وأسعارها هي القيم Values. جدول التجزئة Hash Table يقوم بتعيين المفاتيح Keys إلى القيم Values.

في القسم التالي، سترى بعض الأمثلة حيث تكون جداول التجزئة Hash Tables مفيدة حقًا.

التمارين Exercises

من المهم أن تقوم دوال التجزئة Hash Functions بإرجاع Return نفس المخرجات Output بشكل متسق Consistently لنفس المدخلات Input. إذا لم يفعلوا ذلك، فلن تتمكن من العثور على العنصر Item الخاص بك بعد وضعه في جدول التجزئة Hash Table!

أي من دوال التجزئة Hash Functions التالية متسقة Consistent؟

5.1 `f(x) = 1`

5.2 `f(x) = rand()`

5.3 `f(x) = next_empty_slot()`

5.4 `f(x) = len(x)`

تُرجع "1" لجميع المدخلات Input

تُرجع رقمًا عشوائيًا في كل مرة

تُرجع فهرس الخانة الفارغة التالية في جدول التجزئة

يستخدم طول سلسلة الحروف كفهرس

حالات الاستخدام Use Cases

تُستخدَم جداول التجزئة Hash Tables في كل مكان. سيوضح لك هذا القسم بعض حالات الاستخدام Use Cases.

استخدام جداول التجزئة لعمليات البحث Lookups

يحتوي هاتفك على دليل هاتف مدمج Built-In Phonebook عملي.

كل اسم Name له رقم هاتف Phone Number مرتبط به Associated.

لنفترض أنك تريد إنشاء دليل هاتف مثل هذا. تقوم بتعيين Map أسماء Names الأشخاص إلى أرقام الهاتف Phone Numbers. يحتاج دليل الهاتف الخاص بك إلى هذه الوظائف Functionality:

- إضافة Adding اسم الشخص ورقم الهاتف المرتبط بهذا الشخص.
- ادخال Entering اسم الشخص، والحصول على رقم الهاتف المرتبط بهذا الاسم.

هذه حالة استخدام Use Case مثالية لجداول التجزئة Hash Tables! جداول التجزئة رائعة عندما تريد

- إنشاء تعيين Mapping من شيء إلى شيء آخر
- البحث عن Look Up شيء

من السهل جدًا إنشاء دليل هاتف Building Phone Book. أولاً، قم بعمل جدول تجزئة جديد New Hash Table:

```
>>> phone_book = dict()
```

بالمناسبة، لدى لغة بايثون Python اختصار Shortcut لإنشاء جدول تجزئة جديد New Hash Table. يمكنك استخدام اثنين من الأقواس المعقوفة Two Curly Braces:

```
>>> phone_book = {}
```

مثل `phone_book = dict`

دعنا نقوم بإضافة Add أرقام هواتف بعض الأشخاص إلى دليل الهاتف هذا:

```
>>> phone_book["jenny"] = 8675309
>>> phone_book["emergency"] = 911
```

هذا كل ما في الامر! الآن، لنفترض أنك تريد العثور على رقم هاتف JENNY. فقط قم بتمرير Pass المفتاح Key إلى التجزئة Hash:

```
>>> print phone_book["jenny"]
8675309
```

رقم الهاتف

تخيل لو كان عليك القيام بذلك باستخدام مصفوفة Array بدلاً من ذلك. كيف يمكنك أن تفعل ذلك؟ جداول التجزئة Hash Tables تقوم بتسهيل نمذجة Model علاقة Relationship من عنصر Item إلى آخر.

تُستخدم جداول التجزئة Hash Tables لعمليات البحث Lookups على نطاق Scale أكبر بكثير. على سبيل المثال، لنفترض أنك انتقلت إلى موقع ويب Website مثل http://google.com. يجب أن يقوم جهاز الكمبيوتر الخاص بك بترجمة google.com إلى عنوان IP Address.

لأي موقع ويب Website تذهب إليه، يجب ترجمة العنوان Address إلى عنوان IP Address.

رائع، تعيين Mapping عنوان ويب Web Address إلى عنوان IP Address؟ يبدو أنه حالة استخدام Use Case مثالية لجداول التجزئة Hash Tables! هذه العملية Process تسمى DNS Resolution. جداول التجزئة (هاش) Hash Tables هي إحدى الطرق لتوفير هذه الوظائف Functionality.

منع الإدخالات المتكررة Duplicate Entries

لنفترض أنك تدير حجرة تصويت Voting Booth. بطبيعة الحال، يمكن لكل شخص التصويت مرة واحدة فقط. كيف تتأكد من أنهم لم يقوموا بالتصويت من قبل؟ عندما يأتي شخص ما للتصويت، فإنك تسأل عن اسمه الكامل. ثم تقوم بالتحقق منه في قائمة الأشخاص الذين صوتوا بالفعل.

إذا كان اسمه مدرجًا في القائمة، فهذا يعني أن هذا الشخص قد قام بالتصويت بالفعل، وإذا كان غير مدرج في القائمة، يمكنك إضافة اسمه إلى القائمة والسماح له بالتصويت. لنفترض الآن أن الكثير من الأشخاص قد حضروا للتصويت، وأن قائمة الأشخاص الذين صوتوا طويلة جداً.

في كل مرة يأتي شخص جديد للتصويت، عليك تصفح هذه القائمة العملاقة لمعرفة ما إذا كان قد قام بالتصويت بالفعل. ولكن هناك طريقة أفضل: استخدام التجزئة (هاش) Hash!

أولاً، قم بعمل تجزئة Hash لتتبع Track الأشخاص الذين قاموا بالتصويت:

```
>>> voted = {}
```

عندما يأتي شخص جديد للتصويت، تحقق مما إذا كان موجوداً بالفعل في التجزئة Hash:

```
>>> value = voted.get("tom")
```

الدالة `get` تقوم بإرجاع Return القيمة Value إذا كان "tom" موجودًا في جدول التجزئة Hash Table. وإلا فإنها تقوم بإرجاع `None`. يمكنك استخدام هذا للتحقق مما إذا كان شخص ما قد قام بالتصويت بالفعل!

ها هو الكود Code:

```
voted = {}

def check_voter(name):
  if voted.get(name):
    print "kick them out!"
  else:
    voted[name] = True
    print "let them vote!"
```

دعونا نختبرها Test عدة مرات:

```
>>> check_voter("tom")
let them vote!
>>> check_voter("mike")
let them vote!
>>> check_voter("mike")
kick them out!
```

في المرة الأولى التي يدخل فيها Tom، سيتم طباعة Print عبارة "let them vote!" ثم يدخل Mike، وسيتم طباعة، "let them vote!" ثم يحاول Mike الذهاب مرة أخرى، وسيتم طباعة، "kick them out!".

تذكّر، إذا كنت تقوم بتخزين Store هذه الأسماء في قائمة الأشخاص الذين قاموا بالتصويت، فستصبح هذه الدالة Function في النهاية بطيئة للغاية، لأنه سيتعين عليها إجراء بحث بسيط Simple Search على القائمة بأكملها. لكنك تقوم بتخزين أسمائهم في جدول تجزئة Hash Table بدلاً من ذلك، ويخبرك جدول التجزئة على الفور ما إذا كان اسم هذا الشخص مدرجًا في جدول التجزئة أم لا. التحقق Check من التكرارات Duplicates سريع جدًا باستخدام جدول التجزئة Hash Table.

استخدام جداول التجزئة Hash Tables كذاكرة تخزين مؤقت Cache

حالة استخدام Use Case أخيرة: التخزين المؤقت Caching. إذا كنت تقوم بعمل موقع ويب Website، فربما تكون قد سمعت عن التخزين المؤقت Caching من قبل كأمر جيد يجب القيام به. ها هي الفكرة. لنفترض أنك قمت بزيارة موقع facebook.com:

1. أنت تقدم طلبًا Request إلى خادم فيسبوك Facebook Server.
2. يفكر الخادم Server لمدة ثانية ويأتي بصفحة الويب Web Page لإرسالها إليك.
3. تحصل على صفحة ويب Web Page.

على سبيل المثال، على موقع Facebook، قد يقوم الخادم Server بتجميع Collect كل أنشطة أصدقائك ليقوم بإظهارها لك. يستغرق الأمر بضع ثوانٍ لجمع كل هذه الأنشطة وإظهارها لك. يمكن أن تبدو تلك الثواني وكأنها

وقت طويل بالنسبة للمستخدم User. قد تفكر، "لماذا Facebook بطيء جدًا؟" من ناحية أخرى، يجب أن تخدم خوادم Facebook ملايين الأشخاص، وهذه الثواني تمثّل لهم الكثير. تعمل خوادم Facebook بجدية لخدمة كل تلك المواقع Websites. هل هناك طريقة لجعل Facebook أسرع وجعل خوادمه Servers تقوم بعمل أقل في نفس الوقت؟

افترض أن لديك ابنة أخت تسألك باستمرار عن الكواكب. "كم يبعد المريخ عن الأرض؟" "كم يبعد القمر؟" "كم يبعد كوكب المشتري؟" في كل مرة، عليك أن تجري بحثًا على Google وتعطيها إجابة. تستغرق بضع دقائق. الآن، افترض أنها تسأل دائمًا، "كم يبعد القمر؟" قريبًا جدًا، ستحفظ أن القمر يبعد 238,900 ميل. لن تضطر إلى البحث عن ذلك على Google ... ما عليك سوى تذكّر الإجابة. هذه هي الطريقة التي يعمل بها التخزين المؤقت Caching: تتذكر مواقع الويب Websites البيانات Data بدلاً من إعادة حسابها Recalculating.

إذا قمت بتسجيل الدخول Log In إلى Facebook، فإن كل المحتوى Content الذي تراه مصمم خصيصًا لك. في كل مرة تذهب فيها إلى facebook.com، يتعين على خوادمه Servers التفكير في المحتوى الذي تهتم به. ولكن إذا لم تقم بتسجيل الدخول إلى Facebook، فسترى صفحة تسجيل الدخول Login Page. يرى الجميع نفس صفحة تسجيل الدخول. يُطلب من Facebook نفس الشيء مرارًا وتكرارًا: "أعطني الصفحة الرئيسية Home Page عندما أقوم بتسجيل الخروج Log Out" لذلك فقد توقف عن جعل الخادم Server يقوم بعمل ما لمعرفة شكل الصفحة الرئيسيةHome Page . بدلاً من ذلك، يحفظ شكل الصفحة الرئيسية ويرسلها إليك.

وهذا ما يسمى التخزين المؤقت Caching. له ميزتان:

- تحصل على صفحة الويب Web Page بشكل أسرع، تمامًا كما هو الحال عندما تحفظ المسافة من الأرض إلى القمر. في المرة القادمة التي تسألك فيها ابنة أختك، لن تضطر إلى البحث في Google. يمكنك الرد على الفور.
- على Facebook القيام بعمل أقل.

التخزين المؤقت Caching طريقة شائعة لجعل الأمور أسرع. جميع المواقع Websites الكبيرة تستخدم التخزين المؤقت. و هذه البيانات Data يتم تخزينها مؤقتًا Cached في تجزئة (هاش) Hash!

لا يقتصر Facebook فحسب على التخزين المؤقت Caching للصفحة الرئيسية Home Page. إنه يقوم أيضًا بتخزين مؤقت Caching لصفحة "حول" About Page وصفحة الاتصال Contact Page وصفحة البنود والشروط Terms And Conditions Page وغير ذلك الكثير. لذلك يحتاج إلى تعيين Map من عنوان URL الصفحة إلى بيانات الصفحة Page Data.

عندما تزور صفحة Page على Facebook، فإنها تتحقق Check أولاً مما إذا كانت هذه الصفحة مخزَّنة Stored في التجزئة Hash.

ها هو التخزين المؤقت Caching في الكود Code:

```
cache = {}

def get_page(url):
  if cache.get(url):
    return cache[url]
  else:
    data = get_data_from_server(url)
    cache[url] = data
    return data
```

إرجاع البيانات المخزنة مؤقتًا

يحفظ هذه البيانات أولا في التخزين المؤقت

هنا، يمكنك جعل الخادم Server يقوم بالعمل فقط إذا لم يكن عنوان URL في التخزين المؤقت Cache.
قبل إعادة Return البيانات Data، تقوم بحفظها Save في التخزين المؤقت Cache. في المرة التالية التي يطلب Request فيها شخص ما عنوان الصفحة URL هذا، يمكنك إرسال البيانات من التخزين المؤقت Cache بدلاً من جعل الخادم Server يقوم بالعمل.

الخلاصة

للتلخيص، تعتبر التجزئات Hashes مفيدة لـ

- نمذجة Modeling العلاقات Relationships من شيء إلى شيء آخر
- تصفية - فلترة Filtering التكرارات Duplicates
- التخزين المؤقت للبيانات Caching / حفظها Memorizing بدلاً من جعل خادمك Server يقوم بالعمل

التصادمات Collisions

كما قلت سابقًا، معظم اللغات لديها جداول تجزئة Hash Tables. لست بحاجة إلى معرفة كيفية كتابة جداولك الخاصة. لذلك، لن أتحدث كثيرًا عن التفاصيل الداخلية Internals لجداول التجزئة Hash Tables. لكنك ما زلت تهتم بالأداء Performance! لفهم أداء جداول التجزئة، تحتاج أولاً إلى فهم ماهية التصادمات Collisions.
يغطي القسمان التاليان التصادمات Collisions والأداء Performance.
أولاً، لقد كنت أقول لك كذبة بيضاء. أخبرتك أن دالة التجزئة Hash Function تقوم دائمًا بتعيين Map المفاتيح المختلفة Different Keys إلى خانات مختلفة Different Slots في المصفوفة Array.

في الواقع، يكاد يكون من المستحيل كتابة دالة تجزئة Hash Function تقوم بذلك. لنأخذ مثالاً بسيطًا.
افترض أن المصفوفة Array الخاصة بك تحتوي على 26 خانة Slot.

ودالة التجزئة Hash Function الخاصة بك بسيطة حقًا: فهي تقوم بتخصيص Assign مكانًا Spot في المصفوفة بشكل أبجدي Alphabetically.

ربما يمكنك بالفعل رؤية المشكلة. تريد وضع سعر التفاح APPLES في التجزئة Hash. يتم تخصيص Assign الخانة Slot الأولى لك.

ثم تريد وضع سعر الموز BANANAS في التجزئة Hash. يتم تخصيص Assign الخانة Slot الثانية لك.

كل شيء يسير على ما يرام! لكنك الآن تريد وضع سعر الأفوكادو AVOCADOS في التجزئة Hash الخاصة بك. يتم تخصيص Assign الخانة Slot الأولي لك مرة أخري.

للأسف! التفاح لديه هذه الخانة Slot بالفعل! ماذا نفعل؟ يسمى هذا بالتصادم Collision: تم تخصيص نفس الخانة Slot لمفتاحين Two Keys. هذه مشكلة. إذا قمت بتخزين Store سعر الأفوكادو في تلك الخانة Slot، فستستبدل Overwrite سعر التفاح. ثم في المرة القادمة التي يسأل فيها شخص ما عن سعر التفاح، سيحصل على سعر الأفوكادو بدلاً من ذلك! التصادمات Collisions سيئة، وتحتاج إلى التغلب عليها. هناك العديد من الطرق المختلفة للتعامل مع التصادمات Collisions. أبسط واحد هو هذا: إذا تم تعيين Map مفاتيح متعددة Multiple Key لنفس الخانة Slot، فابدأ قائمة مرتبطة Linked List في تلك الخانة Slot.

في هذا المثال، يتم تعيين Map كل من "APPLE" و "AVOCADO" إلى نفس الخانة Slot. لذلك تبدأ قائمة مرتبطة Linked List في تلك الخانة Slot. إذا كنت تريد معرفة سعر الموز، فلا يزال الأمر سريعًا. إذا كنت تريد معرفة سعر التفاح، فهو أبطأ قليلاً. يجب عليك البحث في هذه القائمة المرتبطة Linked List للعثور على "APPLE". إذا كانت القائمة المرتبطة Linked List صغيرة، فلا مشكلة كبيرة - عليك البحث من خلال ثلاثة أو أربعة عناصر Elements. لكن لنفترض أنك تعمل في محل بقالة حيث تبيع فقط منتجات تبدأ بالحرف A.

مهلا، انتظر لحظة! جدول التجزئة Hash Table بأكمله فارغ Empty تمامًا باستثناء خانة Slot واحدة. وهذه الخانة Slot لديها قائمة مرتبطة Linked List عملاقة! كل عنصر في جدول التجزئة Hash Table هذا موجود في القائمة المرتبطة Linked List. هذا سيء مثل وضع كل شيء في قائمة مرتبطة Linked List لتبدأ بها. سيؤدي ذلك إلى إبطاء جدول التجزئة Hash Table الخاص بك.

يوجد درسان هنا Two Lessons:

- دالة التجزئة Hash Function الخاصة بك مهمة حقًا. دالة التجزئة الخاصة بك قامت بتعيين Map جميع المفاتيح Keys إلي خانة Slot واحدة. بشكل مثالي، يجب أن تعمل دالة التجزئة Hash Function الخاصة بك على تعيين Map المفاتيح Keys بالتساوي Evenly في جميع أنحاء التجزئة Hash.
- إذا أصبحت هذه القوائم المرتبطة Linked Lists طويلة، فإنها تبطئ جدول التجزئة Hash Table كثيرًا. لكن لن تصبح هذه القوائم المرتبطة طويلة إذا كنت تستخدم دالة تجزئة Hash Function جيدة!

دوال التجزئة Hash Functions مهمة. ستمنحك دالة التجزئة Hash Function الجيدة عددًا قليلاً جدًا من التصادمات Collisions. إذن كيف تختار دالة تجزئة Hash Function جيدة؟ هذا هو القادم في القسم التالي!

Performance الأداء

لقد بدأت هذا الفصل في محل البقالة. كنت ترغب في بناء شيء من شأنه أن يمنحك أسعار المنتجات على الفور. حسنًا، جداول التجزئة Hash Tables سريعة حقًا.

في الحالة المتوسطة Average Case، تأخذ جداول التجزئة Hash Tables وقت O(1) لكل شيء.

O(1) يسمى الوقت الثابت Constant Time. لم ترَ وقتًا ثابتًا Constant Time من قبل. لا يعني ذلك فوري - لحظي Instant. هذا يعني أن الوقت المستغرق سيبقى كما هو، بغض النظر عن حجم جدول التجزئة Hash Table. على سبيل المثال، أنت تعلم أن البحث البسيط Simple Search يستغرق وقتًا خطيًا Linear Time.

البحث الثنائي Binary Search أسرع - يستغرق وقت لوغاريتمي Log Time:

يستغرق البحث عن شيء ما في جدول التجزئة Hash Table وقتًا ثابتًا Constant Time.

انظر كيف هو خط مسطح Flat Line؟ هذا يعني أنه لا يهم ما إذا كان جدول التجزئة Hash Table يحتوي على عنصر واحد أو مليار عنصر - سيستغرق الحصول على شيء من جدول التجزئة Hash Table نفس القدر من الوقت. في الواقع، لقد رأيت وقتًا ثابتًا Constant Time من قبل. يستغرق إخراج عنصر Item من مصفوفة Array وقتًا ثابتًا Constant Time. لا يهم حجم المصفوفة الخاصة بك؛ يستغرق الأمر نفس القدر من الوقت للحصول على عنصر. في الحالة المتوسطة Average Case، تكون جداول التجزئة Hash Tables سريعة حقًا. في أسوأ حالة Worst Case، يأخذ جدول التجزئة Hash Table وقت O(n) - الوقت الخطي Linear Time - لكل شيء، وهو حقًا بطيء. دعنا نقارن جداول التجزئة Hash Tables بالمصفوفات Arrays والقوائم Lists.

		جداول التجزئة الحالة المتوسطة HASH TABLES (AVERAGE)	جداول التجزئة أسوأ حالة HASH TABLES (WORST)	المصفوفات ARRAYS	القوائم المرتبطة LINKED LISTS
البحث	SEARCH	O(1)	O(n)	O(1)	O(n)
الإدراج	INSERT	O(1)	O(n)	O(n)	O(1)
الحذف	DELETE	O(1)	O(n)	O(n)	O(1)

انظر إلى الحالة المتوسطة Average Case لجداول التجزئة Hash Tables. تكون جداول التجزئة Hash Tables سريعة مثل المصفوفات Arrays في البحث Searching (الحصول على قيمة Value عند فهرس Index معين). وجداول التجزئة هي بنفس سرعة القوائم المرتبطة Linked Lists في عمليات الإدراج Inserts والحذف Deletes. إنها أفضل ما في العالمين! ولكن في أسوأ حالة Worst Case، تكون جداول التجزئة Hash Tables بطيئة في كل هذا. لذلك من المهم ألا تصل إلى أداءً أسوأ حالة Worst-Case Performance مع جداول التجزئة Hash Tables. وللقيام بذلك، تحتاج إلى تجنب التصادمات Collisions. لتجنب التصادمات، تحتاج إلي

- عامل تحميل منخفض Low Load Factor
- دالة تجزئة Hash Function جيدة

ملاحظة

قبل أن تبدأ في القسم التالي، اعلم أن هذا ليس مطلوبًا للقراءة. سأتحدث عن كيفية تنفيذ Implement جدول التجزئة Hash Table، ولكن لن تضطر أبدًا إلى القيام بذلك بنفسك. مهما كانت لغة البرمجة Programming Language التي تستخدمها، فسيكون لها تنفيذ مدمج Built-In Implementation لجداول التجزئة Hash Tables. يمكنك استخدام جدول التجزئة المدمج Built-In Hash Table وافتراض أنه سيكون له أداء Performance جيد. يمنحك القسم التالي نظرة خاطفة خلف الكواليس.

عامل التحميل Load Factor

من السهل حساب عامل التحميل Load Factor لجدول التجزئة Hash Table.

تستخدم جداول التجزئة Hash Tables مصفوفة Array للتخزين Storage، لذا يمكنك حساب عدد الخانات المشغولة Occupied Slots في مصفوفة Array. على سبيل المثال، يحتوي جدول التجزئة Hash Table هذا على عامل تحميل 2/5 أو 0.4.

ما هو عامل التحميل Load Factor لجدول التجزئة Hash Table هذا؟

إذا قلت 1/3، فأنت على حق. يقيس Measure عامل التحميل Load Factor عدد الخانات الفارغة Empty Slots المتبقية في جدول التجزئة Hash Table.

لنفترض أنك بحاجة إلى تخزين سعر 100 عنصر من المنتجات في جدول التجزئة الخاص بك، وأن جدول التجزئة يحتوي على 100 خانة Slot. في أفضل حالة Best Case، سيحصل كل عنصر على خانة خاصة به.

جدول التجزئة هذا لديه عامل تحميل 1. ماذا لو كان جدول التجزئة يحتوي على 50 خانة فقط؟ عند ذلك يكون عامل التحميل 2. ليس هناك من طريقة أن يحصل كل عنصر على خانة خاصة به، لأنه لا توجد خانات كافية! وجود عامل تحميل Load Factor أكبر من 1 يعني أن لديك عناصر أكثر من الخانات في مصفوفتك. بمجرد أن يبدأ عامل التحميل في النمو Grow، ستحتاج إلى إضافة المزيد من الخانات إلى جدول التجزئة Hash Table الخاص بك. هذا يسمى تغيير الحجم Resizing. على سبيل المثال، افترض أن لديك جدول التجزئة ويتجه أن يكون ممتلئ جدًا.

تحتاج إلى تغيير حجم Resize جدول التجزئة هذا. أولاً تقوم بإنشاء مصفوفة جديدة أكبر. القاعدة الأساسية Rule Of Thumb هي إنشاء مصفوفة ضعف الحجم Twice Size.

أنت الآن بحاجة إلى إعادة إدراج Re-Insert كل هذه العناصر Items في جدول التجزئة الجديد New Hash Table هذا باستخدام دالة hash:

هذا الجدول الجديد له عامل تحميل 3/8. أفضل بكثيرا! مع عامل تحميل أقل Lower Load Factor، سيكون لديك تصادمات Collisions أقل، وسيكون أداء Performance جدولك أفضل. هناك قاعدة أساسية جيدة تتمثل في تغيير الحجم Resize عندما يكون عامل التحميل Load Factor أكبر من 0.7.

قد تفكر، "يستغرق هذا العمل لتغيير الحجم Resize الكثير من الوقت!" وأنت على حق. تغيير الحجم Resizing مكلف، ولا تريد تغيير الحجم كثيرًا. ولكن في المتوسط Average، تأخذ جداول التجزئة وقت $O(1)$ حتى مع تغيير الحجم Resizing.

دالة تجزئة Hash Function جيدة

دالة التجزئة Hash Function الجيدة تقوم بتوزيع Distribute القيم Values في المصفوفة Array بالتساوي Evenly.

دالة التجزئة السيئة تقوم بتجميع Group القيم معًا وتقوم بإنتاج الكثير من التصادمات Collisions.

ما هي دالة التجزئة Hash Function الجيدة؟ هذا شيء لن تقلق بشأنه أبدًا - كبار السن من الرجال (والنساء) ذوي اللحى الكبيرة يجلسون في غرف مظلمة ويقلقون بشأن ذلك. إذا كنت فضولياً حقًا، فابحث عن دالة SHA (يوجد وصف موجز لها في الفصل الأخير). يمكنك استخدام ذلك كدالة التجزئة Hash Function الخاصة بك.

التمارين Exercises

من المهم أن يكون لدوال التجزئة Hash Functions توزيع Distribution جيد. يجب عليها تعيين Map العناصر Items على نطاق واسع Broadly قدر الإمكان. أسوأ حالة Worst Case هي دالة التجزئة Hash Function التي تقوم بتعيين Map جميع العناصر إلى نفس الخانة Slot في جدول التجزئة.

افترض أن لديك دوال التجزئة Hash Functions الأربع هذه التي تعمل مع سلاسل الحروف Strings:

1. تقوم بإرجاع Return القيمة "1" لجميع المدخلات Inputs.
2. تقوم باستخدام طول Length سلسلة الحروف String كفهرس Index.
3. تقوم باستخدام الحرف الأول First Character من سلسلة الحروف String كفهرس Index. لذلك، يتم تجزئة Hashed جميع سلاسل الحروف Strings التي تبدأ بحرف a معًا، وهكذا.
4. تقوم بتعيين Map كل حرف Letter إلى رقم أوَّلي prime number: a = 2 ، b = 3 ، c = 5 ، d = 7 ، e = 11 ، وهكذا. بالنسبة لسلسلة الحروف String، فإن دالة التجزئة Hash Function هي مجموع Sum جميع الأحرف Characters مقسوما قسمة من نوع Modulo (قسمة تعطي الباقي) على حجم التجزئة Hash Size. على سبيل المثال، إذا كان حجم التجزئة الخاص بك هو 10، وسلسلة الحروف String هي "bag"، فإن الفهرس Index هو $(3 + 2 + 17) \% 10 = 22 \% 10 = 2$.

لكل من هذه الأمثلة، ما هي دوال التجزئة Hash Functions التي ستقدِّم توزيعًا Distribution جيدًا؟ افترض أن حجم جدول التجزئة Hash Table Size هو 10 خانات Slots.

5.5 دليل هاتف Phonebook حيث تكون المفاتيح Keys عبارة عن أسماء Names والقيم Values هي أرقام هواتف Phone Numbers. الأسماء كالتالي: Esther و Ben و Bob و Dan.

5.6 تعيين Mapping من حجم البطارية Battery Size إلى الطاقة Power. الأحجام Sizes هي A و AA و AAA و AAAA.

5.7 تعيين Mapping من عناوين الكتب Book Titles إلي المؤلفين Authors. العناوين Titles هي Maus و Fun Home و Watchmen.

الخلاصة

لن تضطر أبدًا إلى تنفيذ Implement جدول التجزئة Hash Table بنفسك. يجب أن توفر لغة البرمجة Programming Language التي تستخدمها التنفيذ Implementation لك. يمكنك استخدام جداول تجزئة للغة بايثون Python وقم بافتراض أنك ستحصل على أداء الحالة المتوسطة Average Case Performance: الوقت الثابت Constant Time.

جداول التجزئة Hash Tables هي هيكل بيانات قوي Powerful Data Structure لأنها سريعة جدًا وتتيح لك نمذجة Model البيانات Data بطريقة مختلفة. قد تكتشف قريبًا أنك تستخدمها طوال الوقت:

- يمكنك عمل جدول تجزئة Hash Table عن طريق دمج Combining دالة تجزئة Hash Function مع مصفوفة Array.
- التصادمات Collisions سيئة. أنت بحاجة إلى دالة تجزئة Hash Function تقلل من التصادمات إلى الحد الأدنى Minimize Collisions.
- جداول التجزئة Hash Tables لديها بحث Search وإدراج Insert وحذف Delete سريع حقًا.
- جداول التجزئة Hash Tables جيدة لنمذجة Modeling العلاقات Relationships من عنصر Item إلى آخر.
- بمجرد أن يكون عامل التحميل Load Factor أكبر من 0.7، فقد حان الوقت لتغيير حجم Resize جدول التجزئة Hash Table.
- تُستخدم جداول التجزئة Hash Tables للتخزين المؤقت للبيانات Caching Data (على سبيل المثال، مع خادم ويب Web Server).
- جداول التجزئة Hash Tables رائعة لاكتشاف التكرارات Duplicates.

6 بحث الاتساع-أولاً
Breadth-First Search

في هذا الفصل

- تتعلم كيفية نمذجة Model شبكة Network باستخدام هيكل بيانات مُجرَّد جديد New Abstract Data Structure: الرسوم البيانية Graphs.
- تتعلم بحث الاتساع-أولا Breadth-First Search، وهو خوارزمية Algorithm يمكنك تشغيلها Run على الرسوم البيانية Graphs للإجابة على أسئلة مثل، "ما هو أقصر مسار للانتقال إلى X؟"
- تتعلم المزيد عن الرسوم البيانية المُوجَّهة Directed Graphs في مقابل الرسوم البيانية غير المُوجَّهة Undirected Graphs.
- تتعلم الترتيب الطوبولوجي Topological Sort، وهو نوع مختلف من خوارزميات الترتيب Sorting Algorithms التي تكشف التبعيات - الاعتمادات Dependencies بين العُقَد Nodes.

يقدم هذا الفصل الرسوم البيانية Graphs. أولاً، سأتحدث عما هي الرسوم البيانية Graphs (لا تشتمل على محور X أو Y). ثم سأعرض لك خوارزمية الرسم البياني Graph Algorithm الأولى. تُسمَّى بحث الاتساع-أولاً Breadth-First Search (BFS).

يتيح لك بحث الاتساع-أولاً Breadth-First Search العثور على أقصر مسافة Shortest Distance بين شيئين. لكن أقصر مسافة Shortest Distance يمكن أن تعني الكثير من الأشياء! يمكنك استخدام بحث الاتساع أولًا Breadth-First Search لـ

- كتابة برنامج ذكاء اصطناعي AI للعبة الداما Checkers التي تحسب أقل عدد من التحركات Fewest Moves لتحقيق النصر
- كتابة مدقِّق إملائي Spell Checker (أقل عدد من التعديلات Fewest Edits من الأخطاء الإملائية إلى كلمة حقيقية - على سبيل المثال، READER -> READED هو أحد التعديلات Edits)
- العثور علي الطبيب الأقرب إليك في شبكتك Network

تعد خوارزميات الرسم البياني Graph Algorithms من أكثر الخوارزميات المفيدة التي أعرفها. تأكد من قراءة الفصول القليلة التالية بعناية - فهذه خوارزميات Algorithms ستتمكن من تطبيقها Apply مرارًا وتكرارًا.

مقدمة إلى الرسوم البيانية Graphs

لنفترض أنك في سان فرانسيسكو San Francisco وتريد الانتقال من Twin Peaks إلى Golden Gate Bridge. تريد الوصول إلى هناك بالحافلة، بأقل عدد ممكن من الانتقالات Transfers Minimum Number. فيما يلي خياراتك.

ما هي خوارزميتك للعثور على المسار Path صاحب أقل عدد من الخطوات Fewest Steps؟

حسنًا، هل يمكنك الوصول إلى هناك في خطوة واحدة؟ فيما يلي جميع الأماكن التي يمكنك الوصول إليها في خطوة واحدة.

لم يتم تسليط الضوء Highlight على الجسر Bridge. لا يمكنك الوصول إلى هناك في خطوة واحدة. هل يمكنك الوصول إلى هناك في خطوتين؟

مرة أخرى، الجسر Bridge غير موجود، لذا لا يمكنك الوصول إلى الجسر في خطوتين. ماذا عن ثلاث خطوات؟

آها! الآن يظهر جسر البوابة الذهبية Golden Gate Bridge. لذلك يستغرق الأمر ثلاث خطوات للانتقال من Twin Peaks إلى الجسر Bridge باستخدام هذا الطريق Route.

هناك طرق Routes أخرى ستوصلك إلى الجسر Bridge أيضًا، لكنها أطول (أربع خطوات). اكتشفت الخوارزمية Algorithm أن أقصر طريق Shortest Route إلي الجسر يبلغ طوله ثلاث خطوات. هذا النوع من المسائل Problems يسمى مسألة أقصر مسار Shortest Path Problem. أنت تحاول دائمًا العثور على أقصر Shortest شيء. قد يكون أقصر طريق إلى منزل صديقك. قد يكون هذا هو أقل عدد من الحركات للفوز Checkmate في لعبة الشطرنج Chess. الخوارزمية المستخدمة لحل مسألة أقصر مسار Shortest-Path Problem تسمى بحث الاتساع-أولاً Breadth-First Search.

لمعرفة كيفية الانتقال من Twin Peaks إلى Golden Gate Bridge، هناك خطوتان:

1. نمذجة Model المسألة Problem كرسم بياني Graph.
2. حل المسألة Problem باستخدام بحث الاتساع-أولاً Breadth-First Search.

بعد ذلك سنغطي ما هي الرسوم البيانية Graphs. ثم سأذهب خلال بحث الاتساع-أولاً Breadth-First Search بمزيد من التفاصيل.

ما هو الرسم البياني Graph؟

يقوم الرسم البياني بنمذجة Model مجموعة من الروابط (العلاقات) Connections. على سبيل المثال، افترض أنك وأصدقاؤك تلعبون لعبة الأوراق، وتريد أن تصمم نموذجًا Model من يدين لمن بالمال. هكذا يمكنك أن تقول، "Alex مدين ل Rama بالمال".

يمكن أن يبدو الرسم البياني الكامل Full Graph شيئًا كهذا.

رسم بياني Graph للأشخاص الذين يدينون بالمال لأشخاص آخرين

Alex مدين ل Rama بالمال، وTom مدين لـ Adit، وهكذا. يتكون كل رسم بياني Graph من عُقَد Nodes وحواف Edges.

هذا كل ما في الامر! تتكون الرسوم البيانية Graphs من العقد Nodes والحواف Edges. العقدة Node يمكن أن ترتبط Connect مباشرة بالعديد من العقد Nodes الأخرى. هذه العقد تسمى جيرانها Neighbors. في هذا الرسم البياني Graph، Rama هو جار Alex. Adit ليس من جيران Alex، لأنهم غير مرتبطين Connected بشكل مباشر. لكن Adit هو جار Rama وTom .

الرسوم البيانية Graphs هي طريقة لنمذجة Model كيفية ارتباط Connection أشياء مختلفة ببعضها البعض.

الآن دعونا نرى بحث الاتساع-أولاً Breadth-First Search أثناء العمل In Action.

بحث الاتساع-أولاً Breadth-First Search

نظرنا إلى خوارزمية بحث Search Algorithm في الفصل الأول: البحث الثنائي Binary Search. بحث الاتساع-أولاً Breadth-First Search هو نوع مختلف من خوارزميات البحث Search Algorithm: خوارزمية يتم تشغيلها Run على الرسوم البيانية Graphs. يمكنها أن تساعد في الإجابة على نوعين من الأسئلة:

- النوع الأول من الأسئلة: هل هناك مسار Path من العقدة A إلى العقدة B؟
- النوع الثاني من الأسئلة: ما هو أقصر مسار Shortest Path من العقدة A إلى العقدة B؟

لقد رأيت بالفعل بحث الاتساع-أولاً Breadth-First Search مرة واحدة، عندما قمت بحساب أقصر طريق من Twin Peaks إلى Golden Gate Bridge. كان هذا سؤالاً من النوع الثاني: "ما هو أقصر مسار Shortest Path؟" الآن دعونا نلقي نظرة على الخوارزمية Algorithm بمزيد من التفصيل. سنطرح سؤالاً من النوع الأول: "هل هناك مسار Path؟"

لنفترض أنك المالك لمزرعة مانجو. كنت تبحث عن بائع مانجو يمكنه بيع المانجو لك. هل أنت على علاقة Connected ببائع مانجو على موقع فيسبوك Facebook؟ حسنًا، يمكنك البحث من خلال أصدقائك.

هذا البحث واضح ومباشر Straightforward.

أولاً، قم بعمل قائمة List بالأصدقاء للبحث خلالها.

الآن، انتقل إلى كل شخص في القائمة List وتحقق Check مما إذا كان هذا الشخص يبيع المانجو.

افترض أن لا أحد من أصدقائك بائع مانجو. الآن عليك البحث من خلال أصدقاء أصدقائك.

في كل مرة تبحث فيها عن شخص ما من القائمة، قم بإضافة Add جميع أصدقائه إلى القائمة List.

بهذه الطريقة، لا تقوم فقط بالبحث خلال أصدقائك، ولكنك تبحث خلال أصدقائهم أيضًا. تذكر أن الهدف هو العثور على بائع مانجو واحد في شبكتك Network. لذا، إذا لم تكن Alice بائعة مانجو، فإنك تضيف صديقاتها إلى القائمة List أيضًا. وهذا يعني أنك ستقوم في النهاية بالبحث خلال أصدقائها — ثم أصدقائهم، وهكذا. باستخدام هذه الخوارزمية Algorithm، ستبحث خلال شبكتك Network بالكامل حتى تصادف بائعًا مانجو. هذه الخوارزمية هي بحث الاتساع-أولاً Breadth-First Search.

إيجاد أقصر مسار Shortest Path

للتلخيص، هذان هما السؤالان اللذان يمكن لبحث الاتساع-أولاً Breadth-First Search الإجابة عليهما:

- النوع الأول من الأسئلة: هل هناك مسار Path من العقدة A إلى العقدة B؟ (مثل: هل يوجد بائع مانجو في شبكتك Network؟)
- النوع الثاني من الأسئلة: ما هو أقصر مسار Shortest Path من العقدة A إلى العقدة B؟ (مثل: من هو أقرب بائع مانجو؟)

رأيت كيف تجيب على السؤال الأول؛ دعنا الآن نحاول الإجابة على السؤال الثاني. هل يمكنك العثور على أقرب بائع مانجو؟ على سبيل المثال، أصدقاؤك هم روابط - علاقات من الدرجة الأولى First-Degree Connections، وأصدقاؤهم هم روابط - علاقات من الدرجة الثانية Second-Degree Connections.

تقوم بتفضيل رابط من الدرجة الأولى First-Degree Connection على رابط من الدرجة الثانية Second-Degree Connection، ورابط من الدرجة الثانية على رابط من الدرجة الثالثة، وهكذا. لذلك يجب ألا تبحث خلال أي معارف من الدرجة الثانية قبل أن تتأكد من عدم وجود رابط بينك وبين بائعي المانجو من الدرجة الأولى. حسنًا، بحث الاتساع-أولاً Breadth-First Search يقوم بهذا بالفعل! الطريقة التي يعمل بها بحث الاتساع-أولاً، هي أن البحث يَتشعَّب إلى الخارج Radiate Out من نقطة البداية Starting Point. لذلك سوف تقوم بالتحقق Check من العلاقات - الروابط من الدرجة الأولى قبل روابط الدرجة الثانية. اختبار مفاجئ: من الذي سيتم التحقق منه أولاً، Claire أم Anuj؟ الإجابة: Claire هي علاقة من الدرجة الأولى، وAnuj من الدرجة الثانية. لذلك سيتم فحص Claire قبل Anuj.

طريقة أخرى لرؤية ذلك، وهي أن روابط الدرجة الأولى First-Degree Connections يتم إضافتها إلى قائمة البحث Search List قبل الروابط من الدرجة الثانية Second-Degree Connections. ما عليك سوى الانتقال إلى أسفل القائمة والتحقق من الأشخاص لمعرفة ما إذا كان أي منهم بائع مانجو. سيتم البحث خلال صلات – روابط Connections الدرجة الأولى قبل الصلات من الدرجة الثانية، لذلك ستجد بائع المانجو الأقرب إليك. لا يعثر بحث الاتساع-أولاً Breadth-First Search على مسار من A إلى B فحسب، بل يعثر أيضًا على أقصر مسار Shortest Path.

لاحظ أن هذا ينجح فقط إذا بحثت الأشخاص بنفس الترتيب Order الذي تمت إضافتهم Added به. هذا يعني أنه إذا تمت إضافة Claire إلى القائمة List قبل Anuj، فيجب بحث Claire قبل Anuj. ماذا يحدث إذا بحثت Anuj قبل Claire، وكان كلاهما بائعي مانجو؟ حسنًا، Anuj هي جهة اتصال من الدرجة الثانية Second-Degree Contact، وClaire هي جهة اتصال من الدرجة الأولى First-Degree Contact. ينتهي بك الأمر مع بائع مانجو ليس الأقرب إليك في شبكتك Network. لذلك أنت بحاجة إلى بحث الأشخاص بالترتيب الذي تمت إضافتهم Added به. هناك هيكل بيانات Data Structure لهذا: يسمى قائمة (طابور) الانتظار Queue.

قوائم الانتظار Queues

تعمل قائمة (طابور) الانتظار Queue تمامًا كما هي في الحياة الواقعية. افترض أنك وصديقك تصطفان في طابور عند محطة الحافلات. إذا كنت أمامه في طابور الانتظار Queue، عليك ركوب الحافلة أولاً. قائمة الانتظار Queue تعمل بنفس الطريقة. قوائم الانتظار Queues تشبه الدفاتر Stacks. لا يمكنك الوصول Access إلى عناصر عشوائية Random Elements في قائمة الانتظار Queue. بدلاً من ذلك، هناك عمليتان Two Operations فقط، الدخول في الطابور Enqueue والخروج من الطابور Dequeue.

إذا (أدرجت عنصرين في قائمة الانتظار) (Enqueue) في القائمة List، فسيتم (إخراج العنصر الأول من قائمة الانتظار) (Dequeue) الذي قمت بإضافته قبل العنصر الثاني. يمكنك استخدام هذا لقائمة البحث Search List الخاصة بك! سيتم فصل Dequeued الأشخاص الذين تمت إضافتهم إلى القائمة أولاً وبحثهم Searched أولاً. تسمى قائمة الانتظار Queue بهيكل بيانات فيفو FIFO Data Structure وهي ترمز إلى (الوارد أولا – يخرج أولا) (First In, First Out). في المقابل، الدفتر Stack عبارة عن هيكل بيانات ليفو (الوارد آخراً – يخرج أولا) (Last In, First Out).

الآن بعد أن تعرفت على كيفية عمل قائمة الانتظار Queue، دعنا ننفذ Implement بحث الاتساع-أولاً Breadth-First Search!

Exercises التمارين

قم بتشغيل Run خوارزمية بحث الاتساع-أولاً Breadth-First Search على كل من هذه الرسوم البيانية Graphs للعثور على الحل Solution.

6.1 أوجد طول Length أقصر مسار Shortest Path من البداية Start إلى النهاية Finish.

6.2 أوجد طول Length أقصر مسار Shortest Path من "CAB" إلى "BAT".

تنفيذ الرسم البياني Implementing Graph

أولاً، تحتاج إلى تنفيذ Implement الرسم البياني Graph بالكود In Code.

يتكون الرسم البياني Graph من عدة عُقَد Nodes.

وكل عقدة Node متصلة Connected بالعقد المجاورة Neighboring Nodes. كيف تُعبِّر Express عن علاقة Relationship مثل "You -> Bob"؟ لحسن الحظ، أنت تعرف هيكل بيانات Data Structure يتيح لك التعبير عن العلاقات: جدول التجزئة Hash Table!

تذكّر أن جدول التجزئة Hash Table يسمح لك بتعيين Map مفتاح Key إلى قيمة Value. في هذه الحالة، تريد تعيين Map عقدة Node لجميع جيرانها Neighbors.

فيما يلي كيفية كتابتها بلغة بايثون Python:

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
```

لاحظ أنه تم تعيين "you" إلى مصفوفة Array. لذا فإن `graph["you"]` سيمنحك مصفوفة Array من جميع جيران "you".

الرسم البياني Graph هو مجرد مجموعة من العُقَد Nodes والحواف Edges، لذلك هذا هو كل ما تحتاجه للحصول على رسم بياني Graph في بايثون Python. ماذا عن رسم بياني أكبر Bigger Graph مثل هذا؟

ها هو كود بلغة بايثون Python:

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
graph["bob"] = ["anuj", "peggy"]
graph["alice"] = ["peggy"]
graph["claire"] = ["thom", "jonny"]
graph["anuj"] = []
graph["peggy"] = []
graph["thom"] = []
graph["jonny"] = []
```

اختبار مفاجئ: هل يهم أي ترتيب Order تُضيف Add به (ثنائيات المفتاح/القيمة) Key/Value Pairs؟

هل يهم إذا كتبت

```
graph["claire"] = ["thom", "jonny"]
graph["anuj"] = []
```

بدلا من

```
graph["anuj"] = []
graph["claire"] = ["thom", "jonny"]
```

فكر عائداً إلى الفصل السابق. الإجابة: لا يهم. جداول التجزئة Hash Tables ليس لديها ترتيب Ordering، لذلك لا يُهِم الترتيب Order الذي تضيف Add به ثنائيات المفتاح/القيمة Key/Value Pairs.

Anuj وPeggy وThom وJonny ليس لديهم أي جيران Neighbors. لديهم أسهم Arrows تشير إليهم، لكن لا يوجد أسهم Arrows تخرج منهم إلى شخص آخر. يسمى هذا بالرسم البياني المُوجَّه Directed Graph – العلاقة Relationship هي اتجاه واحد One Way فقط. إذن Anuj هو جار Bob، لكن Bob ليس جار Anuj.

لا يحتوي الرسم البياني غير المُوجَّه Undirected Graph على أية أسهم Arrows، وكلتا العقدتين Both Nodes جيران لبعضهما البعض. على سبيل المثال، هذان الرسمين البيانيين Graphs متساويان Equal.

تنفيذ الخوارزمية Implementing Algorithm

للتلخيص، إليك كيف سوف يعمل التنفيذ Implementation.

ملاحظة

عند تحديث قوائم الانتظار Updating Queues، أستخدم المصطلحين Enqueue وDequeue. ستصادف أيضًا المصطلحات Push و Pop . غالبًا ما يكون الدفع Push هو نفس الشيء مثل Enqueue، ودائمًا ما يكون السحب Pop هو نفس الشيء مثل Dequeue.

قم بعمل قائمة انتظار Queue للبدء. في Python، يمكنك استخدام دالة قائمة الانتظار ذات النهايتين Double-Ended Queue Function وهي (deque) لهذا الغرض:

```
from collections import deque
search_queue = deque()
search_queue += graph["you"]
```

يُنشئ قائمة انتظار جديدة

يُضيف كل جيرانك Neighbors إلى قائمة انتظار البحث Search

تذكر أن `graph["you"]` سيعطيك قائمة List بجميع جيرانك Neighbors، مثل `["alice", "bob", "claire"]`.

كل هؤلاء يتم إضافتهم Added إلى قائمة انتظار البحث Search Queue.

دعونا نرى الباقي:

```
while search_queue:
  person = search_queue.popleft()
  if person_is_seller(person):
    print person + " is a mango seller!"
    return True
  else:
    search_queue += graph[person]
return False
```

بينما قائمة الانتظار ليست فارغة ...

يسحب الشخص الأول خارج قائمة الانتظار ...

يتحقق مما إذا كان الشخص بائع مانجو

نعم ، إنهم بائعو مانجو.

لا ، ليسوا كذلك. قم بإضافة جميع أصدقاء هذا الشخص إلى قائمة انتظار البحث Search Queue

إذا وصلت إلى هنا، فلا أحد في قائمة الانتظار Queue كان بائع مانجو

شيء أخير: ما زلت بحاجة إلى دالة `person_is_seller` لإخبارك عندما يكون شخص ما بائعًا للمانجو. فيما يلي هذه الدالة:

```
def person_is_seller(name):
  return name[-1] == 'm'
```

هذه الدالة Function تتحقق Check مما إذا كان اسم الشخص ينتهي بالحرف m. إذا كان الأمر كذلك، فهم بائعو مانجو. نوعاً ما طريقة سخيفة للقيام بذلك، لكنها ستعمل في هذا المثال. الآن دعنا نرى بحث الاتساع-أولاً Breadth-First Search أثناء العمل In Action.

وهكذا. ستستمر الخوارزمية Algorithm حتى حدوث أحد الأمرين

- العثور على بائع المانجو
- أو تصبح قائمة الانتظار Queue فارغة Empty، وفي هذه الحالة لا يوجد بائع مانجو.

Alice و Bob لهما صديق مشترك هو: Peggy. لذلك ستتم إضافة Peggy إلى قائمة الانتظار Queue مرتين:

مرة عند إضافة أصدقاء Alice، ومرة أخرى عند إضافة أصدقاء Bob. سينتهي بك الأمر مع اثنين من Peggy في قائمة انتظار البحث Search Queue.

ولكنك تحتاج إلى التحقق من Peggy مرة واحدة لمعرفة ما إذا كانت بائعة مانجو. إذا قمت بالتحقق منها مرتين، فأنت تقوم بعمل إضافي غير ضروري. لذلك بمجرد بحث شخص ما، يجب وضع علامة Mark على هذا الشخص على أنه تم بحثه Searched وعدم بحثه مرة أخرى.

إذا لم تفعل هذا، فقد ينتهي بك الأمر أيضًا في حلقة لا نهائية Infinite Loop. افترض أن الرسم البياني Graph لبائع المانجو بهذا الشكل.

للبدء، تحتوي قائمة انتظار البحث Search Queue على جميع جيرانك Neighbors.

الآن تتحقق من Peggy. إنها ليست بائعة مانجو، لذا تقوم بإضافة Add جميع جيرانها Neighbors إلى قائمة انتظار البحث Search Queue.

بعد ذلك، تتحقق Check من نفسك. أنت لست بائع مانجو، لذا تقوم بإضافة Add جميع جيرانك Neighbors إلى قائمة انتظار البحث Search Queue.

وهكذا. ستكون هذه حلقة لا نهائية Infinite Loop، لأن قائمة انتظار البحث Search Queue ستستمر في الانتقال منك إلى Peggy.

قبل التحقق Check من شخص ما، من المهم التأكد من أنه لم يتم التحقق منه Checked بالفعل. للقيام بذلك، سوف تحتفظ بقائمة List بالأشخاص الذين قمت بالتحقق منهم بالفعل Already Checked.

فيما يلي الكود النهائي Final Code لبحث الاتساع-أولا Breadth-First Search، مع أخذ ذلك في الحسبان:

```
def search(name):
  search_queue = deque()
  search_queue += graph[name]
  searched = []
  while search_queue:
    person = search_queue.popleft()
    if not person in searched:
      if person_is_seller(person):
        print person + " is a mango
seller!"
        return True
      else:
        search_queue += graph[person]
        searched.append(person)
  return False

search("you")
```

هذه المصفوفة هي الطريقة التي يمكنك من خلالها تتبع الأشخاص الذين بحثتهم من قبل

ابحث هذا الشخص فقط إذا لم تكن قد بحثت عنه بالفعل

يتم وضع علامة علي هذا الشخص أنه تم البحث عنه

حاول تشغيل Running هذا الكود Code بنفسك. ربما تحاول تغيير دالة `person_is_ seller` إلى شيء ذو معني بشكل أكبر، وانظر ما إذا كانت تطبع Print ما تتوقعه.

وقت التشغيل Running Time

إذا كنت تبحث في شبكتك Network بالكامل عن بائع مانجو، فهذا يعني أنك ستقوم باتباع Follow كل حافة Edge (تذكر أن الحافة Edge هي السهم Arrow أو العلاقة – الرابط Connection من شخص إلى آخر). لذا فإن وقت التشغيل Running Time هو على الأقل $O(E)$ حيث E هو عدد الحواف Edges.

يمكنك أيضًا الاحتفاظ بقائمة انتظار Queue لكل شخص للبحث خلالها. تستغرق إضافة Adding شخص واحد إلى قائمة الانتظار Queue وقتًا ثابتًا Constant Time: $O(1)$. القيام بذلك لكل شخص سوف يأخذ وقت إجمالي $O(V)$ حيث V هو عدد الأشخاص. يأخذ بحث الاتساع-أولا Breadth-First Search وقت $O(V + E)$، حيث $V + E$ = عدد الأشخاص + عدد الحواف Edges، وهو يكتب بشكل شائع $O(V + E)$ حيث (V عدد الرؤوس Vertices، E عدد الحواف Edges).

التمرين Exercise

هذا رسم بياني Graph صغير لروتيني الصباحي.

يخبرك أنني لا أستطيع تناول الفطور Eat Breakfast حتى أنظف أسناني Brush Teeth. لذا فإن "Eat Breakfast" يعتمد Depend على "Brush Teeth".

من ناحية أخرى، لا يعتمد الاستحمام Showering على تنظيف أسناني Brushing My Teeth، لأنني أستطيع الاستحمام قبل أن أنظف أسناني. من هذا الرسم البياني Graph، يمكنك عمل قائمة List بالترتيب Order الذي أحتاج أن أتبعه في روتيني الصباحي:

1. استيقظ Wake Up.
2. الاستحمام Shower.
3. تنظيف الأسنان Brush Teeth.
4. تناول الإفطار Eat Breakfast.

لاحظ أنه يمكن تحريك "Shower"، لذا فإن هذه القائمة List صالحة Valid أيضًا:

1. Wake Up
2. Brush Teeth
3. Shower
4. Eat Breakfast

6.3 بالنسبة لهذه القوائم الثلاث، حدد ما إذا كانت كل واحدة صالحة Valid أم غير صالحة Invalid.

6.4 هذا رسم بياني أكبر Larger Graph. قم بعمل قائمة صالحة Valid List لهذا الرسم البياني Graph.

يمكنك القول إن هذه القائمة List مرتبة Sorted بطريقة ما. إذا كانت المهمة A تعتمد على المهمة B، فستظهر المهمة A لاحقًا في القائمة. وهذا ما يسمى بالترتيب الطوبولوجي Topological Sort، وهي طريقة لعمل قائمة مرتبة Ordered List من الرسم البياني Graph. لنفترض أنك تخطط لحفل زفاف ولديك رسم بياني Graph كبير مليء بالمهام Tasks التي يجب القيام بها - ولست متأكدًا من أين تبدأ. يمكنك ترتيب Sort الرسم البياني Graph طوبولوجيًا Topologically والحصول على قائمة List بالمهام التي يجب القيام بها، بالترتيب In Order.

افترض أن لديك شجرة عائلة Family Tree.

هذا رسم بياني Graph، لأن لديك عُقد Nodes (الأشخاص) وحواف Edges. تشير الحواف Edges إلى آباء Parents العقد Nodes. لكن كل الحواف Edges تنزل لأسفل - ليس من المنطقي أن يكون لشجرة العائلة حافة Edge تشير من الأسفل للأعلى! سيكون هذا بلا معنى - أباك لا يمكن أن يكون أب جدك!

هذا يسمى بالشجرة Tree. الشجرة هي نوع خاص Special Type من الرسم البياني Graph، حيث لا توجد حواف Edges تشير للأعلي.

6.5 أي من الرسوم البيانية التالية Graphs عبارة عن أشجار Trees أيضًا؟

الخلاصة

- يخبرك بحث الاتساع-أولاً Breadth-First Search إذا كان هناك مسار Path من A إلى B.
- إذا كان هناك مسار Path، فسيجد بحث الاتساع-أولاً Breadth-First Search أقصر مسار Shortest Path.
- إذا كانت لديك مسألة Problem مثل "اعثر على أقصر X"، فحاول وضع نموذج Modeling لمسألتك على هيئة رسم بياني Graph، واستخدم بحث الاتساع-أولاً Breadth-First Search لحلها.
- الرسم البياني الموجه Directed Graph لديه أسهم Arrows، والعلاقة Relationship تتبع Follow اتجاه السهم Arrow Direction (Rama -> Adit تعني Rama يدين بالمال لـ Adit).
- الرسوم البيانية غير الموجهة Undirected Graphs ليس لديها أسهم Arrows، والعلاقة Relationship تسير في كلا الاتجاهين (Ross - Rachel تعني Ross تقابل مع Rachel وRachel تقابل مع Ross).
- قوائم الانتظار Queues هي من نوع FIFO (الوارد أولاً، يخرج أولًا) (First In, First Out).
- الدفاتر Stacks هي من نوع LIFO (الوارد آخراً، يخرج أولًا) (Last In, First Out).
- تحتاج إلى التحقق Check من الأشخاص بالترتيب Order الذي تمت إضافتهم Added به إلى قائمة البحث Search List، لذلك يجب أن تكون قائمة البحث قائمة انتظار Queue. وإلا فلن تحصل على أقصر مسار Shortest Path.
- بمجرد التحقق Check من شخص ما، تأكد من عدم التحقق منه مرة أخرى. بخلاف ذلك، قد ينتهي بك الأمر في حلقة لا نهائية Infinite Loop.

Coding عربي

7 خـــوارزمـيـة ديـكســتـرا
Dijkstra's Algorithm

في هذا الفصل

- نواصل مناقشة الرسوم البيانية Graphs، ونتعرف على الرسوم البيانية الموزونة - المرجوحة Weighted Graphs: وهي طريقة لتخصيص Assign وزن - ترجيح Weight أكثر أو أقل لبعض الحواف Edges.
- تتعلم خوارزمية ديكسترا Dijkstra Algorithm، والتي تتيح لك الإجابة على سؤال "ما هو أقصر مسار إلى X؟" للرسوم البيانية الموزونة - المرجوحة Weighted Graphs.
- تتعرف على الدورات Cycles في الرسوم البيانية Graphs، حيث لا تنجح خوارزمية ديكسترا Dijkstra Algorithm.

في الفصل السابق، اكتشفت طريقة للانتقال من النقطة A إلى النقطة B.

إنه ليس بالضرورة المسار الأسرع Fastest Path. إنه أقصر مسار Shortest Path، لأنه يحتوي على أقل عدد من المقاطع Segments (ثلاثة مقاطع). لكن لنفترض أنك قمت بإضافة Add أوقات الانتقال إلى تلك المقاطع Segments. الآن ترى أن هناك مسارًا أسرع Faster Path.

لقد استخدمت بحث الاتساع-أولاً Breadth-First Search في الفصل السابق. سيجد لك بحث الاتساع-أولاً المسار Path الذي يحتوي على أقل عدد من المقاطع Segments (الرسم البياني Graph الأول الموضح هنا).

ماذا لو كنت تريد المسار الأسرع بدلاً من ذلك (الرسم البياني Graph الثاني)؟ يمكنك القيام بذلك بأسرع طريقة باستخدام خوارزمية مختلفة تسمى خوارزمية ديكسترا Dijkstra Algorithm.

العمل مع خوارزمية ديكسترا Dijkstra Algorithm

دعونا نرى كيف تعمل الخوارزمية مع هذا الرسم البياني Graph.

كل مقطع Segment له وقت انتقال Travel Time بالدقائق Minutes.

أنت ستستخدم خوارزمية Dijkstra للانتقال من البداية Start إلى النهاية Finish في أقصر وقت ممكن.

إذا قمت بتشغيل Run بحث الاتساع-أولاً Breadth-First Search على هذا الرسم البياني Graph، فستحصل على هذا المسار الأقصر Shortest Path.

لكن هذا المسار يستغرق 7 دقائق. دعونا نرى ما إذا كان يمكنك العثور على مسار يستغرق وقتًا أقل! هناك أربع خطوات لخوارزمية ديكسترا Dijkstra:

1. اعثر على العقدة الأرخص Cheapest Node. هذه هي العقدة Node التي يمكنك الوصول إليها في أقل وقت ممكن.
2. قم بتحديث Update تكاليف Costs جيران Neighbors هذه العقدة Node. سأشرح ما أعنيه بهذا بعد قليل.
3. كرِّر الأمر Repeat حتى تنتهي من ذلك مع كل عقدة Node في الرسم البياني Graph.
4. احسب Calculate المسار النهائي Final Path.

الخطوة 1: ابحث عن أرخص عقدة Cheapest Node. أنت تقف في البداية Start، وتتساءل عما إذا كان يجب عليك الانتقال إلى العقدة A أو العقدة B. ما هو الوقت الذي يستغرقه الوصول إلى كل عقدة؟

الوصول إلى العقدة A يستغرق 6 دقائق والوصول إلى العقدة B يستغرق دقيقتين. باقي العقد، لا نعرفها بعد. نظرًا لأنك لا تعرف الوقت المستغرق للوصول إلى النهاية Finish بعد، فأنت تضع ما لا نهاية Infinity (سترى لماذا قريبًا). العقدة B هي أقرب عقدة Closest Node ... وتبعد دقيقتين.

وقت الوصول للعقدة TIME TO NODE

العقدة NODE	TIME TO NODE
A	6
B	2
FINISH	∞

الخطوة 2: احسب المدة التي يستغرقها الوصول إلى جميع جيران Neighbors العقدة B عن طريق اتباع Following حافة Edge من العقدة B.

مهلاً، لقد وجدت للتو مسارًا أقصر للعقدة A! كان الأمر يستغرق 6 دقائق للوصول إلى العقدة A.

ولكن إذا ذهبت من خلال العقدة B، فهناك مسار يستغرق 5 دقائق فقط!

عندما تجد مسارًا أقصر Shorter Path لجار Neighbor للعقدة B، قم بتحديث Update تكلفته Cost. في هذه الحالة، لقد وجدت

- طريق أقصر إلى A (نزولاً من 6 دقائق إلى 5 دقائق)
- مسار أقصر حتى النهاية Finish (نزولاً من اللانهاية Infinity إلى 7 دقائق)

الخطوة 3: كرِّر الأمر!

الخطوة 1 مرة أخرى: اعثر علي العقدة Node التي تستغرق أقل وقت للوصول إليها. لقد انتهيت من العقدة B، لذا فإن العقدة A لديها أقل تقدير زمني تالي Next Smallest Time Estimate.

الخطوة 2 مرة أخرى: قم بتحديث Update التكاليف Costs لجيران Neighbors العقدة A.

رائع، يستغرق الأمر 6 دقائق للوصول إلى النهاية Finish الآن!

لقد قمت بتشغيل Run خوارزمية ديكسترا Dijkstra لكل عقدة Node (لست بحاجة إلى تشغيلها Run على عقدة النهاية Finish Node). في هذه المرحلة، أنت تعرف

- يستغرق الوصول إلى العقدة B دقيقتين.
- يستغرق الوصول إلى العقدة A خمس دقائق.
- يستغرق الوصول إلى النهاية Finish 6 دقائق.

سأوفِّر الخطوة الأخيرة، احتساب Calculating المسار النهائي Final Path، للقسم التالي. في الوقت الحالي، سأريكم فقط ما هو المسار النهائي.

لم يكن ليعثر بحث الاتساع-أولاً Breadth-First Search على هذا باعتباره أقصر مسار Shortest Path، لأنه يحتوي على ثلاثة مقاطع Segments. وهناك طريق للانتقال من البداية Start إلى النهاية Finish في مقطعين Two Segments.

في الفصل السابق، استخدمت بحث الاتساع-أولاً Breadth-First Search للعثور على أقصر مسار بين نقطتين Two Points. في ذلك الوقت، كان أقصر مسار Shortest Path يعني المسار الذي يحتوي على أقل عدد من المقاطع Fewest Segments. ولكن في خوارزمية Dijkstra، تقوم بتخصيص Assign رقم Number أو وزن - ترجيح Weight لكل مقطع Segment. ثم تعثر خوارزمية Dijkstra على المسار صاحب أصغر وزن - ترجيح إجمالي Smallest Total Weight.

للتلخيص، تتكون خوارزمية ديكسترا Dijkstra من أربع خطوات:

1. اعثر Find على أرخص عقدة Cheapest Node. هذه هي العقدة التي يمكنك الوصول إليها في أقل وقت ممكن.
2. تحقق Check مما إذا كان هناك مسار أرخص Cheaper Path لجيران هذه العقدة. إذا كان الأمر كذلك، فقم بتحديث تكاليفهم Costs.
3. كرِّر الأمر حتى تنتهي من ذلك مع كل عقدة في الرسم البياني.
4. احسب المسار النهائي Final Path. (ستأتي في القسم التالي!)

المصطلحات Terminology

أريد أن أريكم بعض الأمثلة الأخرى لخوارزمية ديكسترا Dijkstra أثناء العمل In Action. لكن اسمحوا لي أولا أن أوضح بعض المصطلحات Terminology.

عند العمل باستخدام خوارزمية ديكسترا Dijkstra، فإن كل حافة Edge في الرسم البياني Graph لها رقم Number مرتبط Associated بها. هذه تسمى الأوزان - الترجيحات Weights.

يسمى الرسم البياني Graph ذو الأوزان - الترجيحات Weights بالرسم البياني الموزون – المرجوح Weighted Graph. يُطلق على الرسم البياني بدون أوزان – ترجيحات اسم الرسم البياني غير الموزون – غير المرجوح Unweighted Graph.

لحساب Calculate أقصر مسار في رسم بياني غير مرجوح – غير موزون Unweighted Graph، استخدم بحث الاتساع-أولاً Breadth-First Search. لحساب أقصر مسار في الرسم البياني الموزون - المرجوح Weighted Graph، استخدم خوارزمية ديكسترا Dijkstra. الرسوم البيانية Graphs يمكن أن يكون لديها أيضًا دورات Cycles. تبدو الدورة Cycle مثل هذا.

هذا يعني أنه يمكنك البدء من عقدة Node، والانتقال حولها، وينتهي بك الأمر عند نفس العقدة Node.

لنفترض أنك تحاول العثور على أقصر مسار في هذا الرسم البياني Graph الذي لديه دورة Cycle.

هل يُعقل أن تقوم باتباع Follow الدورة Cycle؟ حسنًا، يمكنك استخدام المسار Path الذي يتجنب الدورة.

أو يمكنك اتباع Follow الدورة Cycle.

ينتهي بك الأمر عند العقدة A في كلتا الحالتين، لكن الدورة Cycle تضيف Adds وزن – ترجيح Weight أكبر.

يمكنك حتى اتباع الدورة Cycle مرتين إذا أردت.

ولكن في كل مرة تقوم باتباع Follow الدورة Cycle، فإنك فقط تضيف 8 إلى الوزن - الترجيح الإجمالي Total Weight. لذا فإن اتباع الدورة لن يمنحك أبداً أقصر مسار Shortest Path.

أخيرًا، تذكّر محادثتنا حول الرسوم البيانية المُوجَّهة Directed Graphs في مقابل غير الموجهة Undirected من الفصل السادس؟

يعني الرسم البياني غير الموجه Undirected Graph أن كلا العقدتين Both Nodes تشير Point إلى بعضهما البعض. هذه دورة Cycle!

مع الرسم البياني غير الموجه Undirected Graph، تُضيف كل حافة Edge دورة Cycle أخرى. خوارزمية ديكسترا Dijkstra تعمل فقط مع الرسوم البيانية اللادورية (غير دورية) الموجهة Directed Acyclic Graphs، والتي تسمى DAGs للاختصار.

المقايضة للحصول على البيانو
Trading For Piano

كفى مصطلحات Terminology، دعونا نلقي نظرة على مثال آخر! هذا Rama.

يحاول راما Rama مقايضة Trade كتاب موسيقي مقابل الحصول على بيانو Piano.

يقول أليكس Alex: "سأعطيك هذا الملصق Poster مقابل الحصول على كتابك".

"إنه ملصق Poster لفرقتي المفضلة وهي Destroyer. أو سأقدم لك هذا LP النادر لـ Rick Astley مقابل كتابك و5 دولارات أخرى ".

آه، لقد سمعت أن LP لديه أغنية رائعة حقًا،" تقول إيمي Amy.

" سأقايض Trade معك غيتاري Guitar أو مجموعة الطبول Drum Set مقابل الملصق Poster أو LP."

"لقد كنت أنوي الدخول في الغيتار Guitar!" يصيح بيتهوفن Beethoven.

"حسناً، سوف أعطيك البيانو Piano الخاص بي مقابل أي من الأشياء التي تخص إيمي Amy."

رائع! بقليل من المال، يمكن ل Rama أن يقايض في طريقه من كتاب بيانو إلى بيانو حقيقي. الآن يحتاج فقط إلى معرفة كيفية إنفاق أقل مبلغ من المال لإجراء تلك الصفقات Trades. دعونا نرسم بشكل بياني Graph ما تم عرضه عليه Out.

في هذا الرسم البياني Graph، العقد Nodes هي جميع العناصر Items التي يمكن لراما Rama المقايضة Trade عليها. الأوزان - الترجيحات Weights على الحواف Edges هي مقدار المال الذي سيتعين عليه دفعه لإتمام الصفقة. حسنا يمكنه مقايضة الملصق Poster بالجيتار مقابل 30 دولارًا، أو مقايضة LP بالجيتار مقابل 15 دولارًا. كيف سيكتشف Rama المسار Path من الكتاب إلى البيانو حيث ينفق أقل القليل؟ ستنقذنا خوارزمية Dijkstra! تذكّر أن خوارزمية ديكسترا Dijkstra Algorithm لديها أربع خطوات. في هذا المثال، ستنفذ جميع الخطوات الأربع، لذا ستحسب Calculate المسار النهائي Final Path في النهاية أيضًا.

قبل أن تبدأ، تحتاج إلى بعض الإعداد Setup. قم بعمل جدول Table لتكلفة Cost كل عقدة Node. تكلفة العقدة هي تكلفة الوصول إليها.

ستستمر في تحديث Updating هذا الجدول مع استمرار الخوارزمية. لحساب المسار النهائي Final Path، تحتاج أيضًا إلى عمود الآباء Parent Column في هذا الجدول.

سأوضح لك كيف يعمل هذا العمود Column قريبًا. لنبدأ الخوارزمية Algorithm.

الخطوة 1: اعثر على أرخص عقدة Cheapest Node. في هذه الحالة، يكون الملصق Poster هو أرخص صفقة Cheapest Trade بسعر 0 دولار. هل هناك طريقة أرخص للمقايضة من أجل الملصق؟ هذه نقطة مهمة حقًا، لذا فكر فيها. هل يمكنك رؤية سلسلة من الصفقات التي سيحصل فيها راما Rama على الملصق بأقل من 0 دولار؟ أكمل القراءة عندما تكون جاهزًا. الإجابة: لا. نظرًا لأن الملصق هو أرخص عقدة Cheapest Node يمكن لراما Rama الوصول إليها، فلا توجد طريقة لجعلها أرخص بأي حال من الأحوال. إليك طريقة مختلفة للنظر إليها. لنفترض أنك تنتقل من المنزل Home إلى العمل Work.

إذا سلكت المسار Path نحو المدرسة School، فسيستغرق ذلك دقيقتين. إذا سلكت المسار المؤدي إلى المنتزه Park، فسيستغرق ذلك 6 دقائق. هل هناك أي طريقة يمكنك من خلالها السير في الطريق نحو المنتزه، وينتهي بك المطاف في المدرسة، في أقل من دقيقتين؟ إنه مستحيل، لأنه يستغرق أكثر من دقيقتين للوصول إلى الحديقة. من ناحية أخرى، هل يمكنك العثور على مسار أسرع إلي الحديقة؟ نعم.

هذه هي الفكرة الأساسية وراء خوارزمية ديكسترا Dijkstra: انظر إلى أرخص عقدة Cheapest Node على الرسم البياني Graph الخاص بك. لا توجد طريقة أرخص للوصول إلى هذه العقدة Node!

العودة إلى مثال الموسيقى. الملصق Poster هو أرخص صفقة Cheapest Trade.

الخطوة 2: اكتشف المدة التي يستغرقها الوصول إلى جيرانها Neighbors (التكلفة Cost).

لديك أسعار الغيتار الجهير Bass Guitar ومجموعة الطبول Drum Set في الجدول Table. تم تعيين Set قيمتهم Value عند المرور بالملصق Poster، لذلك يتم تعيين الملصق باعتباره أباهم Parent. هذا يعني، للوصول إلى الغيتار الجهير، عليك اتباع Follow الحافة Edge من الملصق Poster، ونفس الشيء بالنسبة للطبول Drums.

الأب PARENT	العقدة NODE	التكلفة COST
BOOK	LP	5
BOOK	POSTER	Ø
POSTER	GUITAR	3Ø
POSTER	DRUMS	35
—	PIANO	∞

نذهب من Poster للوصول إلي العُقد

الخطوة 1 مرة أخرى: LP هي العقدة التالية الأرخص Next Cheapest Node بسعر 5 دولارات.

الخطوة 2 مرة أخرى: قم بتحديث Update قيم Values جميع جيرانها Neighbors.

حسناً، لقد قمت بتحديث سعر كل من الطبول والجيتارا! هذا يعني أنه من الأرخص Cheaper الوصول إلى الطبول والغيتار باتباع الحافة Edge من LP. لذلك قمت بتعيين LP باعتباره الأب الجديد New Parent لكلا الآلتين Both Instruments.

الجيتار الجهير Bass Guitar هو العنصر التالي الأرخص Next Cheapest Item. قم بتحديث Update جيرانه Neighbors.

حسنًا، لديك أخيرًا سعر للبيانو Piano، من خلال مقايضة الجيتار Guitar بالبيانو. لذلك قمت بتعيين Set الجيتار كأب Parent. أخيرًا، العقدة الأخيرة Last Node، مجموعة الطبول Drum Set.

يمكن لراما Rama الحصول على البيانو بسعر أرخص أكثر من خلال مقايضة مجموعة الطبول Drum Set بالبيانو بدلاً من ذلك. لذا فإن أرخص مجموعة من الصفقات ستكلف راما 35 دولارًا.

PARENT	NODE
BOOK	LP
BOOK	POSTER
LP	GUITAR
LP	DRUMS
DRUMS	PIANO

الآن، كما وعدت، أنت بحاجة لمعرفة المسار Path. حتى الآن، أنت تعلم أن أقصر مسار Shortest Path يكلف 35 دولارًا، ولكن كيف يمكنك تحديد المسار؟ لكي تبدأ، انظر إلى الأب Parent بالنسبة إلى البيانو Piano.

البيانو Piano لديه الطبول Drums كأب Parent له. هذا يعني أن راما Rama يقايض الطبول من أجل البيانو. لذلك أنت تقوم باتباع Follow هذه الحافة Edge.

دعونا نرى كيف ستقوم باتباع الحواف. البيانو Piano لديه الطبول Drums كآباء Parents له.

والطبول Drums لها LP كأب Parent لها.

لذا راما Rama سوف يقايض LP من أجل الطبول Drums. وبالطبع، سوف يقايض الكتاب Book من أجل LP. باتباع Following الآباء Parents عكسيا Backward، لديك الآن المسار الكامل Complete Path.

فيما يلي سلسلة الصفقات التي يجب على راما Rama القيام بها.

حتى الآن، كنت أستخدم مصطلح أقصر مسار Shortest Path حرفياً Literally: حساب أقصر مسار بين موقعين أو شخصين. آمل أن يوضح لك هذا المثال أن أقصر مسار لا يجب أن يكون متعلقاً بالمسافة المادية Physical Distance. يمكن أن يتعلق الأمر بتقليل Minimize شيء ما. في هذه الحالة، أراد راما Rama تقليل Minimize المبلغ الذي أنفقه. شكرا خوارزمية ديكسترا Dijkstra!

حواف ذات وزن - ترجيح سالب Negative-Weight Edges

في مثال المقايضة Trading، عرض أليكس Alex مقايضة الكتاب Book مقابل عنصرين Items.

لنفترض أن سارة Sarah تعرض مقايضة LP بالملصق Poster، وستعطي لراما 7 دولارات إضافية. لا يتكلف راما Rama أي شيء لإتمام هذه الصفقة؛ بدلاً من ذلك، يحصل على 7 دولارات. كيف تقوم بإظهار هذا على الرسم البياني Graph؟

الحافة Edge من LP إلى الملصق Poster لها وزن - ترجيح سالب Negative Weight! يحصل راما Rama على 7 دولارات إذا قام بهذه الصفقة. الآن لدى Rama طريقان للوصول إلى Poster.

لذلك من المنطقي القيام بالصفقة الثانية - يستعيد Rama دولارين بهذه الطريقة! الآن، إذا كنت تتذكر، يمكن ل Rama مقايضة Poster بال Drums. هناك مساران Two Paths يمكن أن يسلكهما.

المسار الثاني يكلفه أقل بمقدار 2 دولار، لذا يجب أن يسلك هذا المسار Path، أليس كذلك؟ حسنا خمن ماذا؟ إذا قمت بتشغيل Run خوارزمية ديكسترا Dijkstra على هذا الرسم البياني Graph، فسوف يسلك Rama المسار الخطأ. سوف يأخذ المسار الأطول Longer Path. لا يمكنك استخدام خوارزمية ديكسترا Dijkstra إذا كانت لديك حواف ذات وزن - ترجيح سالب Negative-Weight Edges. تعمل الحواف ذات الوزن السالب Negative-Weight Edges على كسر الخوارزمية Break Algorithm. دعونا نرى ما يحدث عند تشغيل خوارزمية ديكسترا Dijkstra على هذا. أولاً، قم بإنشاء جدول Table التكاليف Costs.

بعد ذلك، ابحث عن العقدة الأقل تكلفة Lowest-Cost Node، وقم بتحديث Update التكاليف Costs لجيرانها Neighbors. في هذه الحالة، يكون Poster هو العقدة الأقل تكلفة. لذلك، وفقًا لخوارزمية ديكسترا Dijkstra، لا توجد طريقة أرخص للوصول إلى Poster من دفع 0 دولار (أنت تعلم أن هذا خطأ!). على أي حال، دعونا نُحدِّث التكاليف لجيرانها.

حسنًا، تبلغ تكلفة الوصول إلي Drums الآن 35 دولارًا.

دعنا نحصل على العقدة الأرخص التالية Next-Cheapest Node التي لم تتم معالجتها Processed بالفعل.

قم بتحديث Update التكاليف Costs لجيرانها Neighbors.

لقد عالجت Processed بالفعل عقدة Node الملصق Poster، لكنك تقوم بتحديث التكلفة لها. هذا **علم أحمر كبير (للتنبيه)**. بمجرد معالجة عقدة Node، فهذا يعني أنه لا توجد طريقة أرخص للوصول إلى تلك العقدة.

لكنك وجدت للتو طريق أرخص للوصول إلي Poster! ليس لدي Drums أي جيران Neighbours، إذاً هذه نهاية الخوارزمية. فيما يلي التكاليف النهائية Final Costs.

يتكلف الحصول على ال Drums 35 دولارًا. أنت تعلم أن هناك مسارًا Path يتكلف 33 دولارًا فقط، لكن خوارزمية ديكسترا Dijkstra لم تعثر عليه. افترضت خوارزمية ديكسترا Dijkstra أنه نظرًا لأنك كنت تعالج Processing عقدة Node الملصق Poster، فلا يوجد طريق أسرع للوصول إلى تلك العقدة. هذا الافتراض ينجح فقط إذا لم يكن لديك حواف ذات وزن - ترجيح سالب Negative-Weight Edges. لذلك لا يمكنك استخدام حواف الوزن السالب Negative-Weight Edges مع خوارزمية ديكسترا Dijkstra. إذا كنت تريد العثور على أقصر مسار Shortest Path في رسم بياني Graph يحتوي على حواف ذات وزن سالب Negative-Weight Edges، فهناك خوارزمية لذلك! إنها تسمى خوارزمية بيلمان-فورد Bellman-Ford. وهي خارج نطاق هذا الكتاب، ولكن يمكنك العثور على بعض الشروحات الرائعة عبر الإنترنت.

التنفيذ Implementation

دعونا نرى كيفية تنفيذ خوارزمية ديكسترا Dijkstra بالكود In Code. هذا هو الرسم البياني Graph الذي سأستخدمه في المثال.

لتكويد Coding هذا المثال، ستحتاج إلى ثلاثة جداول تجزئة Hash Tables.

ستقوم بتحديث Update جداول تجزئة Hash Tables التكاليف Costs و الآباء Parents مع تقدم Progress الخوارزمية Algorithm. أولا، تحتاج إلى تنفيذ الرسم البياني Implement The Graph. ستستخدم جدول تجزئة Hash Table كما فعلت في الفصل السادس:

```
graph = {}
```

في الفصل السابق، قمت بتخزين جميع جيران Neighbors العقدة Node في جدول التجزئة Hash Table، على النحو التالي:

```
graph["you"] = ["alice", "bob", "claire"]
```

لكن هذه المرة، تحتاج إلى تخزين الجيران Neighbors والتكلفة Cost للوصول إلى ذلك الجار Neighbor.

على سبيل المثال، لدى Start اثنان من الجيران A و B.

كيف تقوم بتمثيل Represent أوزان - ترجيحات Weights تلك الحواف Edges؟ لماذا لا تستخدم فقط جدول تجزئة Hash Table آخر؟

```
graph["start"] = {}
graph["start"]["a"] = 6
graph["start"]["b"] = 2
```

لذا فإن `graph["start"]` هو جدول تجزئة Hash Table. يمكنك الحصول على جميع الجيران Neighbors ل Start مثل هذا:

```
>>> print graph["start"].keys()
["a", "b"]
```

هناك حافة Edge من Start إلى A وحافة من Start إلى B. ماذا لو كنت تريد إيجاد أوزان Weights تلك الحواف Edges؟

```
>>> print graph["start"]["a"]
2
>>> print graph["start"]["b"]
6
```

دعنا نضيف Add بقية العقد Nodes وجيرانها Neighbors إلى الرسم البياني Graph:

```
graph["a"] = {}
graph["a"]["fin"] = 1
graph["b"] = {}
graph["b"]["a"] = 3
graph["b"]["fin"] = 5
graph["fin"] = {}
```

عقدة النهاية Finish Node ليس لديها أي جيران Neighbors

جدول تجزئة للرسم البياني الكامل Full Graph Hash Table يبدو هكذا.

بعد ذلك، تحتاج إلى جدول تجزئة Hash Table لتخزين تكاليف كل عقدة Node. تكلفة Cost العقدة Node هي المدة التي يستغرقها الوصول إلى تلك العقدة من البداية Start. أنت تعلم أن الأمر يستغرق دقيقتين من Start إلى B. أنت تعلم أن الأمر يستغرق 6 دقائق للوصول إلى العقدة A (على الرغم من أنك قد تجد مسارًا يستغرق وقتًا أقل). أنت لا تعرف كم من الوقت يستغرق للوصول إلى Finish. إذا كنت لا تعرف التكلفة Cost حتي الآن، فإنك تضع لانهاية Infinity. هل يمكنك تمثيل اللانهاية Infinity في بايثون Python؟ تبين أنه يمكنك:

```
infinity = float("inf")
```

فيما يلي الكود Code لعمل جدول التكاليف Costs Table:

```
infinity = float("inf")
costs = {}
costs["a"] = 6
costs["b"] = 2
costs["fin"] = infinity
```

تحتاج أيضًا إلى جدول تجزئة Hash Table آخر للآباء Parents:

فيما يلي الكود لعمل جدول التجزئة Hash Table للآباء Parents:

```
parents = {}
parents["a"] = "start"
parents["b"] = "start"
parents["fin"] = None
```

أخيرًا، أنت بحاجة إلى مصفوفة Array لتتبع Track جميع العقد Nodes التي قمت بمعالجتها Processed بالفعل، لأنك لست بحاجة إلى معالجة Process عقدة Node أكثر من مرة:

```
Processed = []
```

هذا هو كل الإعداد Setup. الآن دعونا نلقي نظرة على الخوارزمية Algorithm.

سأعرض لك الكود أولاً ثم أقوم بجولة خلاله. ها هو الكود:

```
node = find_lowest_cost_node(costs)
while node is not None:
  cost = costs[node]
  neighbors = graph[node]
  for n in neighbors.keys():
    new_cost = cost + neighbors[n]
    if costs[n] > new_cost:
      costs[n] = new_cost
      parents[n] = node
  processed.append(node)
  node = find_lowest_cost_node(costs)
```

اعثر علي العقدة الأقل تكلفة التي لم تقم بمعالجتها حتى الآن

إذا كنت قد عالجت جميع العقد، فستنتهي حلقة while هذه

قم بالمرور على جميع جيران هذه العقدة. إذا كان الوصول إلى هذا الجار أرخص

من خلال المرور بهذه العقدة ...

... قم بتحديث تكلفة هذه العقدة.

تصبح هذه العقدة الأب الجديد لهذا الجار

ضع علامة على العقدة على أنها تمت معالجتها.

ابحث عن العقدة التالية لمعالجتها ثم قم بعمل Loop

هذه هي خوارزمية ديكسترا Dijkstra's Algorithm بلغة بايثون Python! سأعرض لك كود الدالة Function لاحقًا. أولاً، دعنا نرى كود خوارزمية `find_lowest_cost_node` أثناء العمل In Action.

اعثر على العقدة Node صاحبة أقل تكلفة Lowest Cost.

احصل على التكلفة Cost والجيران Neighbors لتلك العقدة Node.

قم بالمرور Loop على الجيران Neighbors.

كل عقدة Node لها تكلفة Cost. التكلفة هي المدة التي يستغرقها الوصول إلى تلك العقدة من البداية Start.

هنا، أنت تحسب المدة التي سيستغرقها الوصول إلى العقدة A إذا اتخذت مسار Start > العقدة B > العقدة A، بدلاً من مسار Start > العقدة A.

دعونا نقارن Compare هذه التكاليف Costs.

لقد وجدت مسارًا أقصر Shorter Path للعقدة A! قم بتحديث Update التكلفة Cost.

يمر المسار الجديد New Path عبر العقدة B، لذا قم بتعيين B باعتبارها الأب الجديد New Parent.

حسنًا، لقد عدت إلى قمة Top الحلقة Loop. الجار التالي Next Neighbor هو عقدة النهاية Finish Node.

كم من الوقت يستغرق الوصول إلى Finish إذا مررت بالعقدة B؟

تستغرق 7 دقائق. كانت التكلفة السابقة عبارة عن دقائق لا متناهية Infinity Minutes و7 دقائق هي أقل من ذلك.

قم بتعيين Set التكلفة الجديدة New Cost والأب الجديد New Parent لعقدة النهاية Finish Node.

حسنًا، لقد قمت بتحديث تكاليف لجميع جيران العقدة B. قم بتمييزها بعلامة Mark على أنها تمت معالجتها Processed.

اعثر علي العقدة التالية Next Node لمعالجتها Process.

احصل على التكلفة والجيران للعقدة A.

العقدة A لديها جار Neighbor واحد فقط: عقدة النهاية Finish Node.

يستغرق حاليًا الوصول إلى عقدة النهاية 7 دقائق. كم من الوقت سيستغرق الوصول إلى هناك إذا مررت بالعقدة A؟

من الأسرع الوصول إلى Finish من العقدة A! دعونا نُحدِّث Update التكلفة Cost والأب Parent.

بمجرد الانتهاء من معالجة جميع العقد Nodes، تنتهي الخوارزمية. آمل أن تساعدك هذه الجولة في فهم الخوارزمية بشكل أفضل قليلاً. من السهل جدًا العثور على العقدة الأقل تكلفة Lowest-Cost Node باستخدام دالة `find_lowest_cost_node`.

ها هي بالكود In Code:

```
def find_lowest_cost_node(costs):
  lowest_cost = float("inf")
  lowest_cost_node = None
  for node in costs:
    cost = costs[node]
    if cost < lowest_cost and node not in processed:
      lowest_cost = cost
      lowest_cost_node = node
  return lowest_cost_node
```

قم بالمرور خلال كل عقدة

إذا كانت الأقل تكلفة حتى الآن ولم يتم معالجتها بعد ...

... قم بتعيينها لتكون العقدة الجديدة الأقل تكلفة

التمرين Exercise

7.1 في كل من هذه الرسوم البيانية Graphs، ما هو وزن - ترجيح Weight أقصر مسار Shortest Path من البداية Start إلى النهاية Finish؟

الخلاصة

- يتم استخدام بحث الاتساع أولًا Breadth-First Search لحساب أقصر مسار Shortest Path للرسم البياني غير الموزون – المرجوح Unweighted Graph.
- تُستخدم خوارزمية Dijkstra لحساب أقصر مسار Shortest Path للرسم البياني المرجوح - الموزون Weighted Graph.
- تنجح خوارزمية Dijkstra عندما تكون جميع الأوزان - الترجيحات Weights موجبة.
- إذا كانت لديك أوزان سالبة Negative Weights، فاستخدم خوارزمية Bellman-Ford.

8 الخوارزميات الطامعة – الشَّرِهة
Greedy Algorithms

في هذا الفصل

- تتعلم كيفية معالجة المستحيل: المسائل Problems التي ليس لها حل خوارزمي سريع Fast Algorithmic Solution (مسائل NP-Complete Problems).
- تتعلم كيفية تمييز Identify مثل هذه المسائل عندما تراها، لذلك لا تضيع الوقت في محاولة العثور على خوارزمية سريعة لها.
- ستتعرف على خوارزميات التقريب Approximation Algorithms، والتي يمكنك استخدامها لإيجاد حل تقريبي Approximate Solution بسرعة لمسألة NP-Complete.
- تتعلم المزيد عن الإستراتيجية الشَّرِهَة – الطامعة Greedy Strategy، وهي إستراتيجية بسيطة جدًا لحل المسائل Problem-Solving.

مسألة جدولة قاعات الدراسة
Classroom Scheduling Problem

لنفترض أن لديك قاعة دراسة Classroom وتريد عقد Hold أكبر عدد ممكن من الفصول الدراسية Classes فيها. لقد حصلت على قائمة List الفصول الدراسية.

الصف CLASS	البداية START	النهاية END
ART	9 AM	10 AM
ENG	9:30AM	10:30AM
MATH	10 AM	11 AM
C S	10:30AM	11:30AM
MUSIC	11 AM	12 PM

لا يمكنك عقد كل هذه الفصول الدراسية Classes في هذه القاعة، لأن بعضها متداخل Overlap.

تريد أن تعقد أكبر عدد ممكن من الفصول الدراسية Classes في قاعة الدراسة Classroom هذه. كيف تختار مجموعة Set الفصول الدراسية Classes التي ستعقدها، بحيث تحصل على أكبر مجموعة ممكنة من هذه الفصول؟

تبدو مسألة صعبة، أليس كذلك؟ في الواقع، الخوارزمية Algorithm سهلة للغاية، وقد تفاجئك. وإليك كيف تعمل:

1. اختر الفصل الدراسي Class الذي سينتهي في أقرب وقت. هذا هو أول فصل ستعقده في قاعة الدراسة Classroom هذه.
2. الآن، عليك اختيار فصل دراسي Class يبدأ بعد الفصل الأول. مرة أخرى، اختر الفصل الذي سينتهي في أقرب وقت. هذا هو الفصل الثاني الذي ستعقده.

استمر في فعل هذا، وستنتهي بالإجابة! دعونا نجربها. ينتهي فصل الفن Art في أقرب وقت ممكن، الساعة 10:00 صباحًا، لذلك هذا سيكون أحد الفصول Classes التي تختارها.

أنت الآن بحاجة إلى الفصل الدراسي التالي Next Class الذي يبدأ بعد الساعة 10:00 صباحًا وينتهي في أقرب وقت.

فصل اللغة الإنجليزية English خارج الاختيار لأنه يتعارض مع فصل الفن Art، لكن فصل الرياضيات Math يمكن اختياره. أخيرًا، يتعارض فصل علوم الحاسب CS مع فصل الرياضيات Math، لكن فصل الموسيقى Music يمكن اختياره.

إذن هذه هي الفصول الدراسية Classes الثلاثة التي ستعقدها في قاعة الدراسة Classroom هذه.

أخبرني الكثير من الناس أن هذه الخوارزمية Algorithm تبدو سهلة. إنها واضحة للغاية، لذا لا بد أنها خطأ.

لكن هذا هو جمال الخوارزميات الطامعة – الشَّرِهَة Greedy Algorithms: أنها سهلة! الخوارزمية الطامعة – الشَّرِهَة Greedy Algorithm تكون بسيطة: في كل خطوة Step، اختر الحركة المثالية Optimal Move. في هذه الحالة Case، في كل مرة تختار فيها فصلاً دراسياً Class، تختار الفصل الذي سينتهي في أقرب وقت.

بالمصطلحات الفنية In Technical Terms: في كل خطوة Step تختار الحل الأمثل محليًا Locally Optimal Solution، وفي النهاية يتبقى لك الحل الأمثل عالميًا Globally Optimal Solution. صدق أو لا تصدق، هذه الخوارزمية البسيطة Simple Algorithm تجد الحل الأمثل Optimal Solution لمسألة الجدولة Scheduling Problem هذه!

من الواضح أن الخوارزميات الطامعة – الشَّرِهَة Greedy Algorithms لا تنجح دائمًا. لكن من السهل كتابتها!

دعونا نلقي نظرة على مثال آخر.

مسألة حقيبة الظهر Knapsack Problem

لنفترض أنك لص جشع - طمَّاع Greedy Thief. أنت في متجر Store بحقيبة ظهر Knapsack، وهناك كل هذه العناصر Items التي يمكنك سرقتها. لكن يمكنك فقط أن تأخذ ما يمكنك احتواءه داخل حقيبة ظهرك Knapsack. يمكن أن تحمل الحقيبة 35 رطلاً.

أنت تحاول تعظيم Maximize قيمة Value العناصر Items التي تضعها في حقيبتك. ما الخوارزمية Algorithm التي تستخدمها؟

مرة أخرى، الإستراتيجية الطامعة – الشَّرِهَة Greedy Strategy بسيطة جدًا:

1. اختر أغلى شيء سوف يتم احتواءه داخل حقيبتك.
2. اختر الشيء الأغلى التالي الذي سوف يتم احتواءه داخل حقيبتك. وهكذا.

باستثناء هذه المرة، فإنها لا تنجح! على سبيل المثال، افترض أن هناك ثلاثة عناصر يمكنك سرقتها.

يمكن لحقيبتك أن تستوعب 35 رطلاً من العناصر. نظام الاستريو Stereo System هو الأغلى ثمناً لذلك تسرقه. الآن ليس لديك مساحة Space لأي شيء آخر.

لقد حصلت على ما قيمته 3000 دولار من السلع. لكن انتظر! إذا اخترت الكمبيوتر المحمول Laptop والغيتار Guitar بدلاً من ذلك، فقد يكون لديك ما قيمته 3500 دولار من الغنائم!

من الواضح أن استراتيجية الشراهة – الطمع - الجشع Greedy Strategy لا تمنحك الحل الأمثل Optimal Solution هنا. لكنها تجعلك قريبًا جدًا. في الفصل التالي، سأشرح كيفية حساب الحل الصحيح Correct Solution. ولكن إذا كنت لصًا في مركز تسوق، فأنت لا تهتم بالكمال. "جيد جدًا" يكفي.

وإليك هذه القاعدة من هذا المثال الثاني: في بعض الأحيان، الكمال Perfect هو عدو الجيد Good. في بعض الأحيان، كل ما تحتاجه هو خوارزمية Algorithm تحل المسألة بشكل جيد. وهذا هو الدور الذي تتألق فيه الخوارزميات الطامعة – الشَّرِهَة Greedy Algorithms، لأنها سهلة الكتابة وعادة ما تكون قريبة جدًا.

التمارين Exercises

8.1 أنت تعمل لدي شركة أثاث، وعليك شحن الأثاث إلى جميع أنحاء البلاد. تحتاج إلى تعبئة شاحنتك بالصناديق. جميع الصناديق بأحجام مختلفة، وأنت تحاول تعظيم Maximize المساحة التي تستخدمها في كل شاحنة. كيف تختار الصناديق لتعظيم المساحة؟ قم بعمل استراتيجية طامعة – شَرِهَة Greedy Strategy. هل سيوفر لك ذلك الحل الأمثل Optimal Solution؟

8.2 أنت ذاهب إلى أوروبا، ولديك سبعة أيام لترى كل ما تستطيع رؤيته. تقوم بتخصيص Assign قيمة نقطية Point Value لكل عنصر Item (مقدار رغبتك في رؤيته) وتقدير المدة التي يستغرقها. كيف يمكنك تعظيم Maximize إجمالي النقاط Total Points (رؤية كل الأشياء التي تريد حقًا رؤيتها) أثناء إقامتك؟ قم بعمل استراتيجية طامعة – شرِهَة Greedy Strategy. هل سيوفر لك ذلك الحل الأمثل Optimal Solution؟

دعونا نلقي نظرة على مثال أخير. هذا مثال حيث الخوارزميات الطامعة – الشَّرِهَة Greedy Algorithms ضرورية للغاية.

مسألة تغطية المجموعات
The Set-Covering Problem

لنفترض أنك تبدأ عرضًا إذاعيًا Radio Show. تريد الوصول إلى المستمعين في جميع ولايات أمريكا الخمسين. عليك أن تقرر ما هي المحطات Stations التي سيتم العرض عليها للوصول إلى كل هؤلاء المستمعين.

RADIO STATION	AVAILABLE IN
KONE	ID,NV,UT
KTWO	WA,ID,MT
KTHREE	OR,NV,CA
KFOUR	NV,UT
KFIVE	CA,AZ

...etc...

أن تعرض على كل محطة Station يكلفك أموالًا، لذا فأنت تحاول تقليل Minimize عدد المحطات التي ستعرض عليها. لديك قائمة List من المحطات Stations. تُغطِّي كل محطة Station منطقة Region، وهناك تداخل Overlap.

كيف يمكنك اكتشاف أصغر مجموعة Set من المحطات التي يمكنك العرض عليها لتغطية جميع الولايات الخمسين؟ يبدو سهلا، أليس كذلك؟ لقد اتضح أنه صعب للغاية. فيما يلي كيفية القيام بذلك:

1. ضع قائمة بكل مجموعة فرعية Subset محتملة من المحطات Stations. هذا يسمى المجموعة الأُسِّية Power Set. هناك مجموعات فرعية محتملة عددها 2^n.
2. من بين هؤلاء، اختر المجموعة Set التي تحتوي على أقل عدد من المحطات والتي تغطي جميع الولايات الخمسين.

المشكلة هي أن حساب كل مجموعة فرعية محتملة من المحطات يستغرق وقتًا طويلاً. يستغرق الأمر وقت $O(2^n)$، لأن هناك محطات عددها 2^n. من الممكن القيام بذلك إذا كان لديك مجموعة صغيرة من 5 إلى 10 محطات. لكن مع كل الأمثلة هنا، فكر فيما سيحدث إذا كان لديك الكثير من العناصر Items. سيستغرق وقتًا أطول بكثير إذا كان لديك المزيد من المحطات. افترض أنه يمكنك حساب 10 مجموعات فرعية Subsets في الثانية.

لا توجد خوارزمية Algorithm يمكنها حلها بالسرعة الكافية! ما الذي تستطيع القيام به؟

عدد المحطات NUMBER OF STATIONS	الوقت المستغرق TIME TAKEN
5	3.2 sec
10	102.4 sec
32	13.6 years
100	4×10^{21} years

خوارزميات التقريب Approximation Algorithms

الخوارزميات الطامعة – الشَّرِهَة Greedy algorithms ستنقذنا! فيما يلي خوارزمية طامعة – شرِهَة تنتج حل قريب جداً:

1. اختر المحطة Station التي تغطي معظم الولايات التي لم تتم تغطيتها بعد. لا بأس إذا كانت المحطة تغطي بعض الولايات التي تمت تغطيتها بالفعل.
2. كرِّر الأمر Repeat حتى يتم تغطية جميع الولايات.

هذا يسمى خوارزمية التقريب Approximation Algorithm. عندما يستغرق حساب الحل الدقيق Exact Solution وقتًا طويلاً جدا، ستنجح خوارزمية التقريب. يتم الحكم على خوارزميات التقريب Approximation Algorithms من خلال

- مدي سرعتها
- مدى قُربها من الحل الأمثل Optimal Solution

تعد الخوارزميات الطامعة – الشَّرِهَة Greedy Algorithms خيارًا جيدًا لأنها ليست سهلة الوصول إليها فحسب، بل إن بساطتها تعني أنها عادةً تعمل بسرعة أيضًا. في هذه الحالة، تعمل الخوارزمية الطامعة – الشَّرِهَة في وقت $O(n^2)$، حيث n هو عدد محطات الراديو Radio Stations.

دعونا نرى كيف تبدو هذه المسألة Problem بالكود.

كود للإعداد Code For Setup

في هذا المثال، سأستخدم مجموعة فرعية Subset من الولايات والمحطات لإبقاء الأمور بسيطة.

أولاً، قم بعمل قائمة List بالولايات التي تريد تغطيتها:

```
states_needed = set(["mt", "wa", "or", "id", "nv", "ut","ca", "az"])
```

تقوم بتمرير Pass مصفوفة Array، ويتم تحويلها Converted إلى مجموعة Set

لقد استخدمت مجموعة Set لهذا. المجموعة تشبه القائمة List، باستثناء أن كل عنصر Item يمكن أن يظهر مرة واحدة فقط في المجموعة. لا يمكن أن تحتوي المجموعات Sets على تكرارات Duplicates. على سبيل المثال، افترض أن لديك هذه القائمة List:

```
>>> arr = [1, 2, 2, 3, 3, 3]
```

وقمت بتحويلها Converted إلى مجموعة Set:

```
>>> set(arr)
set([1, 2, 3])
```

1 و 2 و 3 تظهر جميعها مرة واحدة فقط في المجموعة Set.

تحتاج أيضًا إلى قائمة المحطات التي تختار منها. اخترت استخدام تجزئة Hash لهذا:

```
stations = {}
stations["kone"] = set(["id", "nv", "ut"])
stations["ktwo"] = set(["wa", "id", "mt"])
stations["kthree"] = set(["or", "nv", "ca"])
stations["kfour"] = set(["nv", "ut"])
stations["kfive"] = set(["ca", "az"])
```

المفاتيح Keys هي أسماء المحطات Station Names، والقيم Values هي الولايات States التي تغطيها. إذن في هذا المثال، تغطي محطة كون Kone Station ولايات (أيداهو Idaho ونيفادا Nevada ويوتا Utah). جميع القيم Values هي مجموعات Sets أيضًا. إن جعل كل شيء مجموعة Set سيجعل حياتك أسهل، كما سترى قريبًا.

أخيرًا، أنت بحاجة إلى شيء ما للاحتفاظ بالمجموعة النهائية من المحطات التي سوف تستخدمها:

```
final_stations = set()
```

حساب الإجابة Calculating The Answer

أنت الآن بحاجة إلى حساب المحطات التي ستستخدمها. ألقِ نظرة على الصورة على اليسار، وانظر إذا ما كان بإمكانك التنبؤ بالمحطات التي يجب عليك استخدامها.

يمكن أن يكون هناك أكثر من حل صحيح. أنت بحاجة إلى المرور بكل محطة واختيار المحطة التي تغطي أكثر الولايات التي لم يتم تغطيتها.

سأطلق على هذه best_station:

```
best_station = None
states_covered = set()
for station, states_for_station in stations.items():
```

States_covered هي مجموعة Set من جميع الولايات التي تغطيها هذه المحطة والتي لم تتم تغطيتها بعد. تسمح لك حلقة (Loop) for بالمرور Loop Over على كل محطة لمعرفة المحطة الأفضل.

دعونا نلقي نظرة على جسم الحلقة for:

```
covered = states_needed & states_for_station
if len(covered) > len(states_covered):
best_station = station
states_covered = covered
```

بناء جملة (نحو) جديد New Syntax! وهذا ما يسمى تقاطع المجموعة Set Intersection

يوجد خط Line ذو مظهر مضحك Funny-Looking هنا:

```
covered = states_needed & states_for_station
```

ماذا يحدث هنا؟

المجموعات Sets

افترض أن لديك مجموعة Set من الفاكهة.

لديك أيضًا مجموعة Set من الخضار.

عندما يكون لديك مجموعتان Two Sets، يمكنك القيام ببعض الأشياء المرحة معهم.

فيما يلي بعض الأشياء التي يمكنك فعلها مع المجموعات Sets.

- اتحاد المجموعات Set Union يعني "الجمع Combine بين كلتا المجموعتين Both Sets".
- تقاطع المجموعات Set Intersection " يعني العثور على العناصر Items التي تظهر في كلتا المجموعتين Both Sets " (في هذه الحالة، الطماطم Tomato فقط).
- يعني فرق المجموعات Set Difference "طرح العناصر Subtract Items الموجودة في مجموعة واحدة من العناصر الموجودة في المجموعة الأخرى."

فمثلا:

```
>>> fruits = set(["avocado", "tomato", "banana"])
>>> vegetables = set(["beets", "carrots", "tomato"])
>>> fruits | vegetables
set(["avocado", "beets", "carrots", "tomato", "banana"])
>>> fruits & vegetables
set(["tomato"])
>>> fruits - vegetables
set(["avocado", "banana"])
>>> vegetables - fruits
```

هذا هو اتحاد المجموعات

هذا هو تقاطع المجموعات

هذا هو فرق المجموعات

ما الذي تعتقد سوف يفعله

للتلخيص:

- المجموعات Sets تشبه القوائم Lists، باستثناء أنه لا يمكن أن تحتوي المجموعات Sets على تكرارات Duplicates.
- يمكنك إجراء بعض العمليات الممتعة على المجموعات Sets، مثل الاتحاد Union والتقاطع Intersection والفرق Difference.

بالعودة إلى الكود

لنعد إلى المثال الأصلي.

هذا تقاطع مجموعات Set Intersection:

```
covered = states_needed & states_for_station
```

covered هي مجموعة من الولايات التي كانت في كل من state_needed و state_for_station. covered هي مجموعة الولايات الغير مغطاة والتي تغطيها هذه المحطة! بعد ذلك، تتحقق مما إذا كانت هذه المحطة تغطي ولايات أكثر من best_station الحالية:

```
if len(covered) > len(states_covered):
  best_station = station
  states_covered = covered
```

إذا كان الأمر كذلك، فهذه المحطة هي best_station الجديدة. أخيرًا، بعد انتهاء حلقة for Loop، يمكنك إضافة best_station إلى القائمة النهائية للمحطات:

```
final_stations.add(best_station)
```

تحتاج أيضًا إلى تحديث States_needed. نظرًا لأن هذه المحطة تغطي بعض الولايات، لم تعد هناك حاجة إلى تلك الولايات:

```
states_needed -= states_covered
```

وتقوم بالمرور Loop حتى تصبح state_needed فارغة Empty. فيما يلي الكود الكامل للحلقة Loop:

```
while states_needed:
  best_station = None
  states_covered = set()
  for station, states in stations.items():
    covered = states_needed & states
    if len(covered) > len(states_covered):
      best_station = station
      states_covered = covered

states_needed -= states_covered
final_stations.add(best_station)
```

أخيرًا، يمكنك طباعة final_stations، ويجب أن ترى هذا:

```
>>> print final_stations
set(['ktwo', 'kthree', 'kone', 'kfive'])
```

هل هذا ما توقعته؟ بدلاً من المحطات 1 و 2 و 3 و 5، كان بإمكانك اختيار المحطات 2 و 3 و 4 و 5. لنقارن وقت تشغيل Run Time الخوارزمية الطامعة – الشَّرِهَة Greedy Algorithm بالخوارزمية الدقيقة Exact Algorithm.

عدد المحطات NUMBER OF STATIONS	وقت الخوارزمية الدقيقة $O(n!)$ EXACT ALGORITHM	وقت الخوارزمية الجشعة $O(n^2)$ GREEDY ALGORITHM
5	3.2 sec	2.5 sec
10	102.4 sec	10 sec
32	13.6 yrs	102.4 sec
100	4×10^{24} yrs	16.67 min

التمارين Exercises

لكل من هذه الخوارزميات، حدد ما إذا كانت خوارزمية طامعة – شرِهَة Greedy Algorithm أم لا.

8.3 الترتيب السريع Quicksort

8.4 بحث الاتساع-أولا Breadth-First Search

8.5 خوارزمية ديكسترا Dijkstra's Algorithm

مسائل NP-Complete

لحل مسألة تغطية المجموعات Set-Covering Problem، كان عليك حساب Calculate كل مجموعة محتملة Possible Set.

ربما تم تذكيرك بمشكلة مندوب المبيعات المتنقل Traveling Salesperson من الفصل الأول. في هذه المسألة Problem، يتعين على مندوب المبيعات زيارة خمس مدن مختلفة.

وهو يحاول معرفة أقصر طريق Shortest Route سيأخذه إلى جميع المدن الخمس. للعثور على أقصر طريق، عليك أولاً حساب كل طريق محتمل.

كم عدد الطرق التي يجب عليك حسابها لخمس مدن؟

مندوب المبيعات المتنقل Traveling Salesperson
خطوة بخطوة Step By Step

لنبدأ بشكل صغير. افترض أن لديك مدينتين فقط. هناك طريقان للاختيار من بينها.

نفس الطريق أم مختلف؟

قد تعتقد أن هذا يجب أن يكون نفس الطريق. بعد كل شيء، أليس الطريق Marin > SF بنفس مسافة SF > Marin؟ ليس بالضرورة. بعض المدن (مثل سان فرانسيسكو San Francisco) بها الكثير من الشوارع ذات الاتجاه الواحد One-Way Streets، لذلك لا يمكنك العودة بالطريقة التي أتيت بها. قد تضطر أيضًا إلى الابتعاد مسافة ميل أو ميلين عن الطريق للعثور على طريق سريع Highway. لذلك فإن هذين الطريقين ليسا بالضرورة متطابقين.

قد تتساءل، "في مسألة مندوب المبيعات المتنقل، هل هناك مدينة معينة تحتاج إلى البدء منها؟" على سبيل المثال، لنفترض أنني مندوب مبيعات متنقل. أعيش في سان فرانسيسكو، وأريد الذهاب إلى أربع مدن أخرى. ستكون سان فرانسيسكو مدينة البداية Start City.

لكن في بعض الأحيان لا يتم تعيين Set مدينة البداية. لنفترض أنك شركة فيديكس FedEx تحاول تسليم طرد إلى منطقة الخليج Bay Area. يتم نقل الطرد جواً من شيكاغو إلى أحد مواقع FedEx الخمسين في منطقة الخليج. ثم سينقل هذا الطرد على شاحنة ستسافر إلى مواقع مختلفة لتوصيل الطرود. إلى أي موقع يجب أن يتم نقله جواً؟ هنا موقع البدء Start Location غير معروف. الأمر متروك لك لحساب المسار الأمثل Optimal Path وموقع البدء Start Location من أجل مندوب المبيعات المتنقل.

وقت التشغيل Running Time لكلا الإصدارين Both Versions هو نفسه. لكنه مثال أسهل إذا لم تكن هناك مدينة بداية محددة، لذلك سأستخدم هذا الإصدار.

مدينتان = طريقان محتملان Two Possible Routes.

3 مدن

لنفترض الآن أنك أضفت مدينة أخرى. كم عدد الطرق المحتملة هناك؟

إذا بدأت في بيركلي Berkeley، فلديك مدينتان أخريان تزورهما.

هناك ستة طرق إجمالية، اثنان لكل مدينة يمكنك البدء من خلالها.

إذن، ثلاث مدن = 6 طرق محتملة.

4 مدن

دعونا نضيف مدينة أخرى، فريمونت Fremont. لنفترض الآن أنك بدأت في Fremont.

هناك 6 طرق محتملة تبدأ من Fremont. وللعجب! إنها تشبه إلى حد كبير الطرق الستة التي حسبتها سابقًا، عندما كان لديك ثلاث مدن فقط. باستثناء الآن جميع الطرق لديها مدينة إضافية، Fremont! هناك نمط Pattern هنا. لنفترض أن لديك أربع مدن، واخترت مدينة البداية، Fremont. هناك ثلاث مدن متبقية. وأنت تعلم أنه إذا كانت هناك ثلاث مدن، فهناك 6 طرق مختلفة للتنقل بين تلك المدن. إذا بدأت في Fremont، فهناك 6 طرق محتملة. يمكنك أيضًا البدء من إحدى المدن الأخرى.

أربع مدن بداية محتملة Four Possible Start Cities، مع 6 طرق محتملة لكل مدينة بداية = 4 * 6 = 24 طريقاً محتملاً.

هل تري نمطاً Pattern ما؟ في كل مرة تضيف مدينة City جديدة، فإنك تزيد من عدد الطرق Routes التي يتعين عليك حسابها.

كم عدد الطرق المحتملة ل 6 مدن؟ إذا خمنت 720، فأنت على حق. وهناك 5,040 لـ 7 مدن، و40,320 لـ 8 مدن.

وهذا ما يسمى دالة المضروب Factorial Function (تذكر القراءة عن هذا في الفصل الثالث؟). لذا 120 = !5.

افترض أن لديك 10 مدن. كم عدد الطرق المحتملة؟ 3,628,800 = !10. عليك أن تحسب أكثر من 3 ملايين طريق محتمل لـ 10 مدن. كما ترى، فإن عدد الطرق المحتملة يصبح كبيرًا بسرعة جداً! هذا هو السبب في أنه من المستحيل حساب الحل "الصحيح" Correct Solution لمسألة مندوب مبيعات المتنقل Traveling-Salesperson Problem إذا كان لديك عدد كبير من المدن.

هناك شيء مشترك بين مسألة مندوب مبيعات المتنقل ومسألة تغطية المجموعات Set-Covering Problem: تقوم بحساب كل حل محتمل Possible Solution واختيار أصغر Smallest/أقصر Shortest حل Solution.

كل واحدة من هذه المسائل تعتبر NP-Complete.

التقريب Approximating

ما هي خوارزمية التقريب Approximation Algorithm الجيدة لمندوب المبيعات المتنقل Traveling Salesperson؟ شيء بسيط يجد مسارًا قصيرًا Short Path. انظر هل يمكنك التوصل إلى إجابة قبل مواصلة القراءة.

المسافة الإجمالية: 71 ميلا. ربما ليس أقصر مسار Shortest Path، لكنه لا يزال قصيرًا جدًا.

فيما يلي الشرح المختصر ل NP-Completeness: تشتهر بعض المسائل Problems بصعوبة حلها. مندوب المبيعات المتنقل Traveling Salesperson ومسألة تغطية المجموعات Set-Covering Problem هما مثالان. يعتقد الكثير من الأشخاص الأذكياء أنه من غير الممكن كتابة خوارزمية Algorithm من شأنها حل هذه المسائل Problems بسرعة.

كيف يمكنك معرفة ما إذا كانت المسألة NP-Complete؟

جونا Jonah يختار اللاعبين لفريقه الخيالي لكرة القدم. لديه قائمة بالقدرات التي يريدها: لاعب وسط جيد، راكض جيد للخلف، جيد في المطر، جيد تحت الضغط Under Pressure، وهكذا. لديه قائمة باللاعبين Players، حيث يحقق كل لاعب بعض القدرات Abilities.

يحتاج جونا Jonah إلى فريق Team يحقق كل القدرات، وحجم الفريق محدود Limited. وفجأة أدرك، "انتظر ثانية". "هذه مسألة تغطية المجموعات Set-Covering Problem!"

يمكن لجونا Jonah استخدام نفس خوارزمية التقريب Approximation Algorithm لإنشاء فريقه:

1. ابحث عن اللاعب الذي يحقق أكبر قدر من القدرات التي لم تتحقق بعد.
2. قم بالتكرار Repeat حتى يحقق الفريق جميع القدرات (أو تنفد المساحة في الفريق).

تظهر مسائل NP-Complete في كل مكان! من الجيد معرفة ما إذا كانت المسألة التي تحاول حلها NP-Complete. في هذه المرحلة، يمكنك التوقف عن محاولة حلها بشكل مثالي Perfectly، وحلها باستخدام خوارزمية تقريبية Approximation Algorithm بدلاً من ذلك. لكن من الصعب معرفة ما إذا كانت المسألة التي تعمل عليها NP-Complete. عادةً ما يكون هناك فرق بسيط جدًا بين مسألة يسهل حلها ومسألة NP-Complete. على سبيل المثال، في الفصول السابقة تحدثت كثيرًا عن أقصر المسارات Shortest Paths. أنت تعرف كيفية حساب أقصر طريق للانتقال من النقطة A إلى النقطة B.

ولكن إذا كنت تريد العثور على أقصر مسار يربط Connect بين عدة نقاط، فهذه مسألة مندوب المبيعات المتنقل Traveling-Salesperson Problem، والتي هي مسألة NP-Complete. الإجابة المختصرة: لا توجد طريقة سهلة لمعرفة ما إذا كانت المسألة التي تعمل عليها NP-Complete. فيما يلي بعض القواعد المتبعة:

- إذا كانت خوارزميتك Algorithm تعمل بسرعة مع عدد قليل من العناصر Items ولكنها تبطئ حقًا مع المزيد من العناصر.
- تشير جملة "جميع تركيبات X" - "(All Combinations Of X)" عادةً إلى مسألة NP-Complete.
- هل يتعين عليك حساب "كل نسخة محتملة" (Every Possible Version) من X لأنه لا يمكنك تقسيمها إلى مسائل فرعية Sub-Problems أصغر؟ قد تكون NP-Complete.
- إذا كانت مسألتك تتضمن تسلسلاً Sequence (مثل تسلسل المدن Cities Sequence، مثل مندوب مبيعات المتنقل Traveling Salesperson)، وكان من الصعب حلها، فقد تكون NP-Complete.
- إذا كانت مسألتك تتضمن مجموعة Set (مثل مجموعة من محطات الراديو) وكان من الصعب حلها، فقد تكون NP-Complete.
- هل يمكنك إعادة صياغة مسألتك على أنها مسألة تغطية المجموعات Set-Covering Problem أو مشكلة مندوب مبيعات المتنقل Traveling-Salesperson Problem؟ إذن مسألتك هي بالتأكيد NP-Complete.

التمارين Exercises

8.6 يحتاج ساعي البريد للتسليم إلى 20 منزلا. يحتاج إلى العثور على أقصر طريق يصل إلى جميع المنازل العشرين. هل هذه مسألة NP-Complete؟

8.7 العثور على أكبر صحبة في مجموعة من الناس (الصحبة هي مجموعة من الأشخاص يعرفون بعضهم البعض). هل هذه مسألة NP-Complete؟

8.8 أنت ترسم خريطة للولايات المتحدة الأمريكية، وتحتاج إلى تلوين الولايات المتجاورة بألوان مختلفة. عليك أن تجد الحد الأدنى لعدد الألوان التي تحتاجها بحيث لا توجد ولايتان متجاورتان لهما نفس اللون. هل هذه مسألة NP-Complete؟

الخلاصة

- الخوارزميات الطامعة – الشَّرِهَة Greedy Algorithms تقوم بالتحسين Optimize محليًا Locally، على أمل أن ينتهي بها الأمر مع المستوى الأمثل عالميًا Global Optimum.
- لا يوجد حل سريع معروف لمسائل NP-Complete.
- إذا كانت لديك مسألة NP-Complete، فإن أفضل رهان لك هو استخدام خوارزمية تقريبية Approximation Algorithm.
- من السهل كتابة الخوارزميات الطامعة – الشَّرِهَة Greedy Algorithms وتشغيلها بسرعة Fast To Run، لذا فهي تشكل خوارزميات تقريب Approximation Algorithms جيدة.

Coding عربي

9 البرمجة الديناميكية
Dynamic Programming

في هذا الفصل

- تتعلم البرمجة الديناميكية Dynamic Programming، وهي أسلوب لحل مسألة Problem صعبة عن طريق تقسيمها إلى مسائل فرعية Subproblems وحل تلك المشكلات الفرعية أولاً.
- باستخدام الأمثلة، تتعلم كيفية التوصل إلى حل برمجة ديناميكي Dynamic Programming Solution لمسألة جديدة New Problem.

مسألة حقيبة الظهر
The Knapsack Problem

دعنا نعيد النظر في مسألة حقيبة الظهر من الفصل الثامن. أنت لص مع حقيبة ظهر يمكن أن تحمل 4 أرطال lbs من السلع.

لديك ثلاثة عناصر يمكنك وضعها في الحقيبة.

ما هي الأشياء التي يجب أن تسرقها حتى تتمكن من سرقة أقصى قيمة Maximum من الأموال من خلال السلع؟

The Simple Solution الحل البسيط

أبسط خوارزمية Simplest Algorithm هي كالتالي: تقوم بتجربة كل مجموعة محتملة Possible Set من السلع واعثر على المجموعة Set التي تمنحك أكبر قيمة Most Value.

هذا ينجح، لكنه بطيء حقًا. لثلاثة عناصر Items، عليك أن تحسب 8 مجموعات محتملة Possible Sets. لأربعة عناصر، عليك حساب 16 مجموعة. مع كل عنصر تقوم بإضافته، يتضاعف Double عدد المجموعات التي ينبغي حسابها Calculate! تستغرق هذه الخوارزمية وقت $O(2^n)$، وهو بطيء جدًا جدًا.

هذا غير عملي لأي عدد معقول من السلع. في الفصل الثامن، رأيت كيفية حساب حل تقريبي Approximate Solution. سيكون هذا الحل قريبًا من الحل الأمثل Optimal Solution، لكنه قد لا يكون الحل الأمثل.

إذن كيف تحسب الحل الأمثل Optimal Solution؟

البرمجة الديناميكية Dynamic Programming

الإجابة: مع البرمجة الديناميكية! دعونا نرى كيف تعمل خوارزمية البرمجة الديناميكية هنا Dynamic Programming Algorithm. تبدأ البرمجة الديناميكية بحل المسائل الفرعية Subproblems وتبني على ذلك لحل المسألة الكبيرة Big Problem.

بالنسبة لمسألة حقيبة الظهر Knapsack Problem، ستبدأ بحل المسألة للحقائب الأصغر Smaller Knapsacks أو الحقائب الفرعية Sub-Knapsacks ثم العمل على حل المسألة الأصلية Original Problem.

البرمجة الديناميكية Dynamic Programming مفهوم صعب Hard Concept، لذلك لا تقلق إذا لم تفهمها على الفور. سنلقي نظرة على الكثير من الأمثلة.

سأبدأ أولاً بعرض الخوارزمية أثناء العمل In Action. بعد رؤيتها أثناء العمل مرة واحدة، سيكون لديك الكثير من الأسئلة! سأبذل قصارى جهدي للإجابة على كل سؤال.

كل خوارزمية برمجة ديناميكية Dynamic-Programming Algorithm تبدأ بشبكة Grid. فيما يلي شبكة Grid لمسألة حقيبة الظهر Knapsack Problem.

صفوف Rows الشبكة Grid هي العناصر Items، والأعمدة Columns عبارة عن أوزان حقيبة الظهر Knapsack Weights من 1 رطل إلى 4 رطل. أنت بحاجة إلى كل هذه الأعمدة Columns لأنها ستساعدك على حساب Calculate قيم Values حقائب الظهر الفرعية Sub-Knapsacks.

تبدأ الشبكة فارغة Empty. ستقوم بملء Fill كل خلية Cell في الشبكة Grid. بمجرد ملء الشبكة، ستحصل على إجابتك لهذه المسألة Problem! يُرجى المتابعة معي. أنشئ شبكتك Grid الخاصة، وسنعمل على ملأها Fill معًا.

صف الجيتار The Guitar Row

سأعرض لك المعادلة الدقيقة Exact Formula لحساب Calculating هذه الشبكة Grid لاحقًا. دعونا نقوم بجولة أولاً. ابدأ بالصف الأول First Row.

هذا هو صف الجيتار Guitar Row، مما يعني أنك تحاول احتواء Fit الجيتار داخل حقيبة الظهر. في كل خلية Cell، هناك قرار Decision بسيط: هل تسرق الجيتار أم لا؟ تذكر أنك تحاول العثور على مجموعة العناصر التي ستسرقها والتي ستمنحك أكبر قيمة Most Value.

الخلية الأولى First Cell لديها حقيبة ظهر Knapsack بسعة 1 رطل lb. يبلغ وزن الجيتار أيضًا 1 رطل، مما يعني أنه يتم احتواءه Fit داخل الحقيبة! إذن قيمة هذه الخلية 1500 دولار، وتحتوي على جيتار Guitar.

لنبدأ في ملء الشبكة Filling In Grid.

على هذا النحو، ستحتوي كل خلية Cell في الشبكة Grid على قائمة List بجميع العناصر Items التي يتم احتواءها Fit داخل الحقيبة في هذه المرحلة.

دعونا نلقي نظرة على الخلية التالية Next Cell. هنا لديك حقيبة بسعة 2 رطل. حسنًا، الجيتار سيتم احتواءه هناك بالتأكيد!

الشيء نفسه بالنسبة لبقية الخلايا Cells في هذا الصف Row. تذكر، هذا هو الصف الأول First Row، لذا ليس لديك سوى الجيتار للاختيار. أنت تتظاهر بأن العنصرين الآخرين غير متاحين للسرقة في الوقت الحالي.

في هذه المرحلة، ربما تكون في حيرة من أمرك. لماذا تفعل هذا مع حقائب الظهر بسعة 1 رطل، 2 رطل، وهكذا، عندما تتحدث المسألة Problem عن حقيبة ظهر 4 أرطال؟ تذكر كيف أخبرتك أن البرمجة الديناميكية Dynamic Programming تبدأ بمسألة صغيرة Small Problem وتبني عليها للوصول إلى المسألة الكبيرة؟ أنت هنا تحل مسائل فرعية Subproblems والتي ستساعدك على حل المسألة الكبيرة Big Problem. تابع القراءة، وستصبح الأمور أكثر وضوحًا.

في هذه المرحلة، يجب أن تبدو شبكتك Grid هكذا.

تذكر أنك تحاول تعظيم Maximize قيمة الحقيبة. هذا الصف Row يُمثِّل Represent أفضل تخمين حالي Current Best Guess لهذا الحد الأقصى Max. لذا في الوقت الحالي، وفقًا لهذا الصف Row، إذا كان لديك حقيبة بسعة 4 أرطال، فإن القيمة القصوى Max Value التي يمكنك وضعها داخلها ستكون 1500 دولار.

أنت تعلم أن هذا ليس الحل النهائي Final Solution. أثناء عملنا مع الخوارزمية Algorithm، ستعمل على تحسين – تنقيح Refine تقديرك Estimate.

صف الاستريو The Stereo Row

دعونا نقوم بالصف التالي Next Row. هذا الصف مخصص لـ Stereo. الآن بعد أن أصبحت في الصف الثاني Second Row، يمكنك سرقة جهاز الاستريو أو الجيتار Guitar. في كل صف، يمكنك سرقة العنصر الموجود في هذا الصف أو العناصر الموجودة في الصفوف فوقه. لذلك لا يمكنك اختيار سرقة الكمبيوتر المحمول Laptop في الوقت الحالي، ولكن يمكنك سرقة جهاز الاستريو و/أو الجيتار. لنبدأ بالخلية الأولى، حقيبة سعتها 1 رطل. القيمة القصوى الحالية Current Max Value التي يمكنك وضعها في حقيبة 1 رطل هي 1500 دولار.

هل يجب أن تسرق الاستريو Stereo أم لا؟

لديك حقيبة بسعة 1 رطل. هل سيتم احتواء Fit الاستريو هناك؟ كلا، إنه ثقيل جدًا! نظرًا لأنه لا يمكنك وضع جهاز Stereo، يظل 1500 دولار هو الحد الأقصى للتخمين Max Guess لحقيبة وزنها 1 رطل.

نفس الشيء بالنسبة للخليتين التاليتين Next Two Cells. تبلغ سعة هذه الحقائب 2 رطل و3 أرطال. وكانت القيمة القصوى القديمة Old Max Value لكليهما 1500 دولار.

لا يزال جهاز Stereo غير ممكن احتواءه، لذلك تظل تخميناتك دون تغيير.

ماذا لو كان لديك حقيبة بسعة 4 أرطال؟ آها: الاستريو يتم احتواءه Fit أخيرًا! كانت القيمة القصوى القديمة 1500 دولار، ولكن إذا وضعت جهاز الاستريو هناك بدلا من ذلك، فستكون القيمة 3000 دولارا! لنأخذ جهاز Stereo.

لقد قمت للتو بتحديث Updated تقديرك Estimate! إذا كان لديك حقيبة ظهر بسعة 4 أرطال، فيمكنك احتواء ما لا يقل عن 3000 دولار من البضائع فيه. يمكنك أن ترى من الشبكة Grid أنك تقوم بتحديث تقديرك بشكل تدريجي Incrementally.

صف الكمبيوتر المحمول The Laptop Row

دعونا نفعل نفس الشيء مع الكمبيوتر المحمول Laptop! يزن الكمبيوتر المحمول 3 أرطال، لذا لن يتم احتواءه داخل حقيبة بحجم 1 رطل أو 2 رطل. يبقى تقدير Estimate أول خليتين عند 1500 دولار.

عند 3 أرطال، كان التقدير القديم 1500 دولار. ولكن يمكنك اختيار الكمبيوتر المحمول Laptop بدلاً من ذلك، وتبلغ قيمته 2000 دولار. لذا فإن التقدير الأقصى الجديد New Max Estimate هو 2000 دولار!

عند 4 أرطال، تصبح الأشياء ممتعة حقًا. هذا جزء مهم. التقدير الحالي Current Estimate هو 3000 دولار. يمكنك وضع Laptop في الحقيبة، لكن ثمنه لا يتجاوز 2000 دولار.

حسنًا، هذا ليس جيدًا مثل التقدير القديم Old Estimate. لكن انتظر! يزن الكمبيوتر المحمول 3 أرطال فقط، وبذلك تحصل على 1 رطل فارغاً! يمكنك وضع شيء في هذا الـ 1 رطل.

ما هي القيمة القصوى Maximum Value التي يمكن أن يتم احتواءها داخل 1 رطل من المساحة Space؟ حسنًا، لقد كنت تحسبها طوال الوقت.

وفقًا لآخر أفضل تقدير Last Best Estimate، يمكنك احتواء Guitar داخل مساحة 1 رطل، وتبلغ قيمته 1500 دولار. لذا فإن المقارنة الحقيقية هي على النحو التالي.

ربما كنت تتساءل عن سبب حساب القيم القصوى Max Values للحقائب الأصغر Smaller Knapsacks.

آمل الآن أن يكون ذلك منطقيًا! عندما يكون لديك مساحة متبقية، يمكنك استخدام الإجابات Answers على تلك المسائل الفرعية Subproblems لمعرفة ما يمكن احتواءه داخل تلك المساحة Space. من الأفضل أن تأخذ الكمبيوتر المحمول Laptop + الجيتار Guitar مقابل 3500 دولار.

الشبكة النهائية Final Grid تبدو هكذا.

ها هي الإجابة: الحد الأقصى للقيمة Maximum Value التي يمكن احتواءها داخل الحقيبة هي 3500 دولار، مكوّنة من جيتار وجهاز كمبيوتر محمول!

ربما تعتقد أنني استخدمت معادلة Formula مختلفة لحساب قيمة تلك الخلية Cell الأخيرة. هذا لأنني تخطيت Skipped بعض التعقيدات غير الضرورية عند ملء قيم الخلايا السابقة. يتم حساب قيمة كل خلية بنفس المعادلة Formula. ها هي.

يمكنك استخدام هذه المعادلة Formula مع كل خلية Cell في هذه الشبكة Grid، ويجب أن ينتهي بك الأمر بنفس الشبكة Grid التي قمت أنا بها. تذكر كيف تحدثت عن حل المسائل الفرعية Solving Subproblems؟ لقد جمعت Combined الحلول لمسألتين فرعيتين لحل المسألة الأكبر.

الأسئلة الشائعة حول مسألة حقيبة الظهر
Knapsack Problem FAQ

ربما لا يزال هذا يبدو وكأنه سحر. هذا القسم يجيب على بعض الأسئلة الشائعة.

ماذا يحدث إذا قمت بإضافة عنصر Add Item؟

لنفترض أنك أدركت أن هناك عنصراً رابعًا يمكنك سرقته ولم تلاحظه من قبل! يمكنك أيضًا سرقة جهاز iPhone.

هل يجب عليك إعادة حساب Recalculate كل شيء لحساب هذا العنصر الجديد New Item؟ لا. تذكر أن البرمجة الديناميكية Dynamic Programming تستمر في البناء بشكل تدريجي Progressively على تقديرك Estimate. حتى الآن، هذه هي القيم القصوى Max Values.

	1	2	3	4
GUITAR	$1500 G	$1500 G	$1500 G	$1500 G
STEREO	$1500 G	$1500 G	$1500 G	$3000 S
LAPTOP	$1500 G	$1500 G	$2000 L	$3500 LG

هذا يعني أنه بالنسبة لحقيبة ظهر بحجم 4 أرطال، يمكنك سرقة ما قيمته 3500 دولار من السلع. كنت تعتقد أن هذه كانت القيمة القصوى النهائية Final Max Value. ولكن دعونا نقوم بإضافة صف Row لجهاز iPhone.

	1	2	3	4
GUITAR	$1500 G	$1500 G	$1500 G	$1500 G
STEREO	$1500 G	$1500 G	$1500 G	$3000 S
LAPTOP	$1500 G	$1500 G	$2000 L	$3500 LG
IPHONE				

إجابة جديدة

ظهر أن لديك قيمة قصوى جديدة New Max Value! حاول ملء هذا الصف الجديد قبل مواصلة القراءة.

لنبدأ بالخلية الأولى. يتم احتواء جهاز iPhone داخل حقيبة 1 رطل. كان الحد الأقصى القديم 1500 دولار، لكن قيمة iPhone تبلغ 2000 دولار. لنأخذ iPhone بدلاً من ذلك.

	1	2	3	4
GUITAR	$1500 G	$1500 G	$1500 G	$1500 G
STEREO	$1500 G	$1500 G	$1500 G	$3000 S
LAPTOP	$1500 G	$1500 G	$2000 L	$3500 LG
IPHONE	$2000 I			

$1500 G	$1500 G	$1500 G	$1500 G
$1500 G	$1500 G	$1500 G	$3000 S
$1500 G	$1500 G	$2000 L	$3500 LG
$2000 I	$3500 IG		

في الخلية التالية Next Cell، يمكنك احتواء iPhone وGuitar.

بالنسبة للخلية رقم 3، لا يمكنك أن تفعل أفضل من أخذ iPhone وGuitar مرة أخرى،
لذا اترك الأمر كما هو.

بالنسبة للخلية الأخيرة Last Cell، تصبح الأشياء ممتعة. الحد الأقصى الحالي Current Max هو 3500 دولار. يمكنك سرقة iPhone بدلاً من ذلك، ولديك 3 أرطال من المساحة المتبقية.

هذه الثلاثة أرطال تساوي 2000 دولار!

2000 دولار من iPhone + 2000 دولار من المشكلة الفرعية القديمة Old Subproblem القديمة: هذه 4000 دولار. حد أقصي جديد New Max!

فيما يلي الشبكة النهائية الجديدة New Final Grid.

سؤال: هل ستنخفض قيمة Value العمود Column في أي وقت؟ هل هذا ممكن؟

فكر في إجابة Answer قبل إكمال القراءة.

الإجابة: لا. في كل إعادة - تكرار Iteration، تقوم بتخزين Storing الحد الأقصى الحالي للتقدير Current Max Estimate. لا يمكن أن يصبح التقدير Estimate أسوأ مما كان عليه من قبل!

التمرين Exercise

9.1 لنفترض أنه يمكنك سرقة عنصر آخر: MP3 player.

يزن 1 رطل ويبلغ سعره 1000 دولار. هل يجب أن تسرقه؟

ماذا يحدث إذا قمت بتغيير ترتيب الصفوف Order of Rows؟

هل الإجابة تتغير؟ لنفترض أنك ملأت الصفوف بهذا الترتيب: Stereo، Laptop، Guitar. كيف تبدو الشبكة Grid؟ املأ الشبكة بنفسك قبل مواصلة القراءة.

هذا ما تبدو عليه الشبكة Grid.

الإجابة لا تتغير. لا يهم ترتيب الصفوف Order of Rows.

هل يمكنك ملء الشبكة Fill Grid من حيث الأعمدة Column-Wise بدلاً من الصفوف Row-Wise؟

جربها بنفسك! بالنسبة لهذه المسألة Problem، فإنها لا تحدث فرقًا. يمكن أن تُحدِث فرقاً لمسائل أخرى.

ماذا يحدث إذا قمت بإضافة عنصر أصغر Add Smaller Item؟

افترض أنه يمكنك سرقة قلادة Necklace. تزن 0.5 رطل وتبلغ قيمتها 1000 دولار. حتى الآن، تفترض شبكتك Grid أن جميع الأوزان Weights أعداد صحيحة Integers. الآن قررت سرقة القلادة. لديك 3.5 رطل متبقية.

ما هي القيمة القصوى Max Value التي يمكنك احتوائها في 3.5 رطل؟ أنت لا تعرف! لقد حسبت فقط قيم Values لحقائب 1 رطل و2 رطل و 3 أرطال و 4 أرطال. أنت بحاجة لمعرفة قيمة حقيبة 3.5 رطل.

بسبب القلادة Necklace، عليك مراعاة التفاصيل الدقيقة Finer Granularity، لذلك يجب تغيير الشبكة Grid.

هل يمكنك سرقة كسور Fractions من عنصر Item؟

لنفترض أنك لص في محل بقالة Grocery Store. يمكنك سرقة أكياس العدس والأرز. إذا كان كيس كامل لا يمكن احتواءه Fit داخل حقيبتك، يمكنك فتحه وأخذ ما يمكنك حمله. إذن الآن ليس كل شيء أو لا شيء - يمكنك أخذ جزء صغير من عنصر ما. كيف تتعامل مع هذا باستخدام البرمجة الديناميكية Dynamic Programming؟

الجواب: لا يمكنك ذلك. مع حل البرمجة الديناميكية Dynamic-Programming Solution، إما أن تأخذ العنصر Item أو لا.

لا توجد طريقة لدي البرمجة الديناميكية لمعرفة أنه يجب عليك أن تأخذ نصف عنصر Half an Item.

لكن هذه الحالة يمكن حلها بسهولة باستخدام خوارزمية شرهة - طامعة Greedy Algorithm! أولاً، خذ أكبر قدر ممكن من العنصر الأكثر قيمة. عندما ينفد ذلك، خذ أكبر قدر ممكن من العنصر التالي الأكثر قيمة، وهكذا.

على سبيل المثال، افترض أن لديك هذه العناصر للاختيار من بينها.

الكينوا Quinoa أغلى من أي شيء آخر من حيث الرطل Pound. لذا، خذ كل الـ Quinoa التي يمكنك حملها! إذا ملأ هذا حقيبتك، فهذا أفضل ما يمكنك فعله.

إذا نفد Quinoa ولا يزال لديك مساحة في حقيبتك، خذ العنصر التالي الأكثر قيمة، وهكذا.

تحسين Optimizing خط سير سفرك

لنفترض أنك ذاهب إلى لندن لقضاء عطلة جميلة. لديك يومان هناك وهناك الكثير من الأشياء التي تريد القيام بها. لا يمكنك فعل كل شيء، لذا عليك عمل قائمة List.

ATTRACTION	TIME	RATING
WESTMINSTER ABBEY	½ DAY	7
GLOBE THEATER	½ DAY	6
NATIONAL GALLERY	1 DAY	9
BRITISH MUSEUM	2 DAYS	9
ST. PAUL'S CATHEDRAL	½ DAY	8

لكل شيء تريد رؤيته، قم بتدوين المدة التي سيستغرقها وتقييم Rate مقدار رغبتك في رؤيته. هل يمكنك معرفة ما يجب أن تراه بناءً على هذه القائمة؟

إنها مسألة حقيبة الظهر Knapsack Problem مرة أخرى! بدلاً من حقيبة الظهر، لديك فترة زمنية محدودة. وبدلاً من Stereo وLaptop، لديك قائمة بالأماكن التي تريد الذهاب إليها. ارسم شبكة البرمجة الديناميكية Dynamic-Programming Grid لهذه القائمة List قبل مواصلة القراءة.

هذا ما تبدو عليه الشبكة Grid.

	½	1	1½	2
WESTMINSTER				
GLOBE THEATRE				
NATIONAL GALLERY				
BRITISH MUSEUM				
ST. PAUL'S				

هل فهمتها جيدا؟ املأ الشبكة Grid. ما الأماكن التي يجب أن ينتهي بك الأمر إلى رؤيتها؟ فيما يلي الإجابة.

التعامل مع العناصر Handling Items التي تعتمد على بعضها البعض

لنفترض أنك تريد الذهاب إلى باريس Paris، لذا أضفت عنصرين إلى القائمة.

EIFFEL TOWER	1½ DAY	8
THE LOUVRE	1½ DAY	9
NOTRE DAME	1½ DAY	7

تستغرق هذه الأماكن الكثير من الوقت، لأنه عليك أولاً السفر من لندن London إلى باريس Paris. هذا يستغرق نصف يوم. إذا كنت تريد القيام بكل الأشياء الثلاثة، فسوف يستغرق الأمر أربعة أيام ونصف.

انتظر، هذا ليس صحيحًا. ليس عليك السفر إلى باريس لكل عنصر. بمجرد وصولك إلى باريس، من المفترض أن يستغرق كل عنصر يومًا واحدًا فقط. لذلك يجب أن يكون يوم واحد لكل عنصر + نصف يوم سفر = 3.5 أيام وليس 4.5 أيام.

إذا وضعت برج إيفل Eiffel Tower في حقيبة الظهر (المسألة)، يصبح متحف اللوفر Louvre "أرخص" - سيكلفك يومًا واحدًا فقط بدلاً من 1.5 يومًا.

كيف تقوم بنمذجة Model هذا في البرمجة الديناميكية Dynamic Programming؟

لا يمكنك. البرمجة الديناميكية قوية لأنها يمكن أن تحل المسائل الفرعية Subproblems وتستخدم تلك الإجابات لحل المسألة الكبيرة. لا تنجح البرمجة الديناميكية إلا عندما تكون كل مسألة فرعية Subproblem منفصلة Discrete – أي أنها لا تعتمد على مسائل فرعية أخرى. هذا يعني أنه لا توجد طريقة لأخذ باريس في الاعتبار باستخدام خوارزمية البرمجة الديناميكية Dynamic-Programming Algorithm.

هل من الممكن أن يتطلب الحل أكثر من حقيبتي ظهر فرعيتين Sub-Knapsacks؟

من الممكن أن الحل الأفضل Best Solution يتضمن سرقة أكثر من عنصرين Two Items. الطريقة التي يتم بها إعداد Set Up الخوارزمية، أنت تجمع Combining بين حقيبتين على الأكثر - لن يكون لديك أكثر من حقيبتين فرعيتين. ولكن من الممكن لتلك الحقائب الفرعية أن يكون لديها حقائبها الفرعية.

هل من الممكن ألا يملأ الحل الأفضل Best Solution حقيبة الظهر Knapsack بالكامل؟

نعم. افترض أنك تستطيع أيضًا سرقة ألماسة Diamond.

هذه ألماسة كبيرة تزن 3.5 أرطال. إنها تساوي مليون دولار، أكثر بكثير من أي شيء آخر. يجب عليك بالتأكيد سرقتها! ولكن لم يتبق سوى نصف رطل من المساحة، ولن يتم احتواء أي شيء في تلك المساحة.

التمرين Exercise

9.2 لنفترض أنك ذاهب للتخييم Camping. لديك حقيبة ظهر Knapsack تستوعب 6 أرطال، ويمكنك أن تأخذ العناصر التالية. لكل منها قيمة Value، وكلما ارتفعت القيمة، زادت أهمية العنصر:

- ماء Water، 3 أرطال، قيمة 10
- كتاب Book، 1 رطل، قيمة 3
- سترة Jacket، 2 رطل، قيمة 5
- كاميرا Camera، 1 رطل، قيمة 6

ما هي المجموعة المثلى Optimal Set من العناصر التي يجب أن تأخذها في رحلة التخييم؟

أطول سلسلة حروف فرعية مشتركة
Longest Common Substring

لقد رأيت مسألة برمجة ديناميكية Dynamic Programming Problem واحدة حتى الآن. ما هي القواعد المتبعة؟

- البرمجة الديناميكية مفيدة عندما تحاول تحسين Optimize شيء ما في ظل قيد Given a Constraint. في مسألة حقيبة الظهر Knapsack Problem، كان عليك تعظيم Maximize قيمة السلع المسروقة، مقيدًا Constrained بحجم الحقيبة.
- يمكنك استخدام البرمجة الديناميكية عندما يمكن تقسيم المسألة إلى مسائل فرعية منفصلة Discrete Subproblems، ولا تعتمد على بعضها البعض.

قد يكون من الصعب التوصل إلى حل برمجة ديناميكية Dynamic-Programming Solution. هذا ما سنركز عليه في هذا القسم. فيما يلي بعض النصائح العامة:

- كل حل برمجة ديناميكية يتضمن شبكة Grid.
- عادةً ما تكون القيم Values الموجودة في الخلايا Cells هي ما تحاول تحسينه Optimize. بالنسبة لمسألة حقيبة الظهر، كانت القيم Values هي قيمة السلع.
- كل خلية Cell هي مشكلة فرعية Subproblem، لذا فكر في كيفية تقسيم مسألتك إلى مسائل فرعية. سيساعدك ذلك على معرفة ما هي المحاور Axes.

دعونا نلقي نظرة على مثال آخر. افترض أنك تقوم بإدارة موقع dictionary.com. يكتب شخص ما كلمة Word، وأنت تعطيه التعريف Definition.

ولكن إذا أخطأ شخص في تهجئة Misspell كلمة ما، فأنت تريد أن تكون قادرًا على تخمين الكلمة التي قصدها. يبحث أليكس Alex عن fish، لكنه كتب hish بدون قصد. هذه ليست كلمة في قاموسك، ولكن لديك قائمة من الكلمات المشابهة Similar Words.

(هذا مثال للتخيل فقط، سوف تقوم بقصر قائمتك على كلمتين. في الواقع، قد تكون هذه القائمة على الأرجح آلاف الكلمات)

Alex كتب hish. ما الكلمة التي قصد أليكس كتابتها: fish أم vista؟

عمل الشبكة Making Grid

كيف تبدو الشبكة Grid الخاصة بهذه المسألة؟ أنت بحاجة للإجابة على هذه الأسئلة:

- ما هي قيم Values الخلايا Cells؟
- كيف تقسم هذه المسألة إلى مسائل فرعية Subproblems؟
- ما هي محاور Axes الشبكة Grid؟

في البرمجة الديناميكية Dynamic Programming، تحاول تعظيم Maximize شيء ما. في هذه الحالة، أنت تحاول العثور على أطول سلسلة حروف فرعية Longest Substring والتي تشترك فيها كلمتان Two Words. ما هي سلسلة الحروف الفرعية Substring التي تشترك فيها hish و fish؟ ماذا عن hish و vista ؟ هذا ما تريد حسابه Calculate.

تذكر أن قيم Values الخلايا Cells هي عادةً ما تحاول تحسينه Optimize. في هذه الحالة، ستكون القيم على الأرجح رقمًا Number: طول Length أطول سلسلة حروف فرعية Longest Substring تشترك فيها السلسلتان Two Strings.

كيف تقسم هذه المسألة إلى مسائل فرعية Subproblems؟ يمكنك مقارنة Compare سلاسل الحروف الفرعية Substrings. بدلاً من المقارنة بين hish و fish، يمكنك مقارنة his و fis أولاً. ستحتوي كل خلية Cell على طول Length أطول سلسلة حروف فرعية Substring التي تشترك فيها سلسلتان فرعيتان Two Substrings. يمنحك هذا أيضًا مؤشراً على أن المحاور Axes ستكون على الأرجح هي الكلمتين Two Words. لذا ربما تبدو الشبكة Grid هكذا.

إذا كان هذا يبدو لك مثل السحر الأسود، فلا تقلق. هذه أشياء صعبة - ولهذا السبب أقوم بتدريسها في مكان متأخر جدًا في الكتاب! لاحقًا، سأعطي لك تمرينًا Exercise حتى تتمكن من ممارسة البرمجة الديناميكية Practice Dynamic Programming بنفسك.

ملء الشبكة Filling Grid

الآن لديك فكرة جيدة عن الشكل الذي يجب أن تبدو عليه الشبكة Grid. ما هي معادلة Formula ملء Filling كل خلية Cell في الشبكة Grid؟ يمكنك الغش قليلاً، لأنك تعرف بالفعل كيف يجب أن يكون الحل - hish وfish لهما سلسلة حروف فرعية Substring مشتركة بطول 3 وهي ish.

لكن ذلك لا يزال لا يخبرك بالمعادلة Formula التي يجب استخدامها. يمزح علماء الكمبيوتر Computer Scientists أحيانًا حول استخدام خوارزمية فينمان Feynman Algorithm. سُميت خوارزمية فينمان على اسم الفيزيائي الشهير ريتشارد فينمان Richard Feynman، وهي تعمل كالتالي:

1. اكتب المسألة.
2. فكر بجدية.
3. اكتب الحل.

علماء الكمبيوتر Computer Scientists هم مجموعة مرحة!

الحقيقة أنه لا توجد طريقة سهلة لحساب المعادلة Formula هنا. عليك أن تجرب وتحاول إيجاد شيء ينجح. أحيانًا لا تكون الخوارزميات Algorithms وصفة دقيقة Exact Recipe. إنها إطار عمل Framework تبني فكرتك فوقه.

حاول التوصل إلى حل لهذه المسألة بنفسك. سأقدم لك تلميحًا — جزء من الشبكة Grid يبدو هكذا.

	H	I	S	H
F	0	0		
I				
S			2	0
H				3

ما هي القيم Values الأخرى؟ تذكر أن كل خلية Cell هي قيمة مسألة فرعية Subproblem Value. لماذا الخلية (3, 3) لها قيمة 2؟ لماذا الخلية (3, 4) لها قيمة 0؟

تابع القراءة بعد أن تكون قد حاولت التوصل إلى معادلة Formula بنفسك. حتى لو لم تفهمها بشكل صحيح، فإن توضيحي سيكون أكثر منطقية.

The Solution الحل

فيما يلي الشبكة النهائية Final Grid.

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	0	0	0	3

فيما يلي المعادلة Formula الخاصة بي لملء كل خلية.

إليك كيف تبدو المعادلة Formula بالكود الزائف Pseudocode:

```
if word_a[i] == word_b[j]:
  cell[i][j] = cell[i-1][j-1] + 1
else:
  cell[i][j] = 0
```

إذا تطابقت الحروف Letters Match

إذا لم تتطابق الحروف Letters Don't Match

هذه هي الشبكة Grid الخاصة بـ hish في مقابل vista.

هناك شيء واحد يجب ملاحظته: بالنسبة لهذه المسألة، قد لا يكون الحل النهائي Final Solution في الخلية الأخيرة Last Cell! بالنسبة لمسألة حقيبة الظهر Knapsack Problem، كانت هذه الخلية الأخيرة دائمًا لديها الحل النهائي. ولكن بالنسبة لأطول سلسلة حروف فرعية مشتركة Longest Common Substring، يكون الحل هو الرقم الأكبر في الشبكة Grid - وقد لا يكون هو الخلية الأخيرة.

دعنا نعود إلى السؤال الأصلي: ما هي سلسلة الحروف String التي لديها مشترك أكثر مع hish؟ تشترك hish و fish في سلسلة حروف فرعية Substring من ثلاثة أحرف مشتركة. تشترك hish و vista في سلسلة فرعية من حرفين مشتركين.

ربما قصد أليكس Alex كتابة fish.

أطول متتالية مشتركة Longest Common Subsequence

لنفترض أن أليكس Alex قد بحث عن طريق الخطأ عن fosh. أي كلمة قصدها: fish أم fort؟

دعنا نقارن بينهما باستخدام معادلة أطول سلسلة حروف فرعية مشتركة Longest-Common-Substring Formula.

كلاهما نفس الشيء: حرفان Two Letters! لكن fosh أقرب إلى fish.

أنت تقارن أطول سلسلة حروف فرعية مشتركة Longest Common Substring، لكنك تحتاج حقًا إلى مقارنة أطول سلسلة حروف متتالية مشتركة Longest Common Subsequence: عدد الأحرف المتتالية Sequence التي تشترك فيها الكلمتان. كيف تحسب أطول متتالية مشتركة Longest Common Subsequence؟

هذه هي الشبكة الجزئية Partial Grid ل fish و fosh.

	F	O	S	H
F	1	1		
I	1			
S		1	2	2
H				

هل يمكنك معرفة المعادلة Formula لهذه الشبكة Grid؟ أطول متتالية - متتابعة مشتركة Longest Common Subsequence تشبه إلى حد بعيد أطول سلسلة حروف فرعية مشتركة Longest Common Substring، والمعادلات Formulas متشابهة جدًا أيضًا. حاول حلها بنفسك - سأعطيك الإجابة بعد ذلك.

أطول متتالية مشتركة – الحل

Longest Common Subsequence—Solution

فيما يلي الشبكة النهائية.

فيما يلي المعادلة Formula الخاصة بي لملء كل خلية.

وها هي بالكود الزائف Pseudocode:

```
if word_a[i] == word_b[j]:
  cell[i][j] = cell[i-1][j-1] + 1
else:
  cell[i][j] = max(cell[i-1][j], cell[i][j-1])
```

إذا تطابقت الحروف

إذا لم تتطابق

يا للعجب - لقد فعلت ذلك! هذا بالتأكيد أحد أصعب فصول الكتاب. إذن، هل تم استخدام البرمجة الديناميكية Dynamic Programming قبل ذلك حقًا؟ نعم:

- يستخدم علماء الأحياء Biologists أطول متتالية مشتركة Longest Common Subsequence لإيجاد أوجه التشابه Similarities في خيوط الحمض النووي DNA Strands. يمكنهم استخدام ذلك لمعرفة مدى تشابه حيوانين Two Animals أو مرضين Two Diseases. يتم استخدام أطول متتالية مشتركة Longest Common Subsequence لإيجاد علاج لمرض التصلب المتعدد Multiple Sclerosis.
- هل سبق لك استخدام Diff (مثل `git diff`)؟ يخبرك Diff بالاختلافات Differences بين ملفين Two Files، ويستخدم البرمجة الديناميكية Dynamic Programming للقيام بذلك.
- تحدثنا عن تشابه سلاسل الحروف String Similarity. تقيس مسافة ليفينشتاين Levenshtein Distance مدى تشابه سلسلتين Two Strings، وتستخدم البرمجة الديناميكية. تُستخدم مسافة Levenshtein في كل شيء بدءًا من التدقيق الإملائي Spell-Check وحتى اكتشاف ما إذا كان مستخدم User يقوم برفع Uploading بيانات محمية بحقوق الطبع والنشر Copyrighted Data.
- هل سبق لك استخدام تطبيق App يقوم بالتفاف الكلمات Word Wrap، مثل Microsoft Word؟ كيف يكتشف مكان الالتفاف بحيث يظل طول الخط متَّسقا Consistent؟ بواسطة البرمجة الديناميكية Dynamic Programming!

Coding عربي

التمرين Exercise

9.3 ارسم الشبكة Grid واملأها لحساب أطول سلسلة حروف فرعية مشتركة Longest Common Substring بين الأزرق Blue والقرائن Clues.

الخلاصة

- البرمجة الديناميكية مفيدة عندما تحاول تحسين Optimize شيء ما في ظل قيد Constraint.
- يمكنك استخدام البرمجة الديناميكية عندما يمكن تقسيم المسألة إلى مسائل فرعية منفصلة Discrete Subproblems.
- كل حل برمجة ديناميكية يتضمن شبكة Grid.
- عادةً ما تكون القيم Values الموجودة في الخلايا Cells هي ما تحاول تحسينه Optimize.
- كل خلية هي مسألة فرعية Subproblem، لذا فكر في كيفية تقسيم مسألتك إلى مسائل فرعية.
- لا توجد معادلة Formula واحدة لحساب حل البرمجة الديناميكية Dynamic-Programming Solution.

Coding عربي

10 K-Nearest Neighbors
أقرب عدد K من الجيران

في هذا الفصل

- تتعلم كيفية بناء نظام تصنيف Classification System باستخدام خوارزمية أقرب عدد K من الجيران K-Nearest Neighbors Algorithm.
- تتعلم عن استخراج الملامح (جمع مَلْمَح) Feature Extraction.
- تتعلم المزيد عن الانحدار Regression: التنبؤ برقم Predicting Number، مثل قيمة سهم Stock Value، أو مقدار استمتاع المستخدم User بفيلم ما.
- تتعرف على حالات الاستخدام Use Cases والمحددات Limitations الخاصة بخوارزمية أقرب عدد K من الجيران K-Nearest Neighbors.

تصنيف البرتقال Oranges في مقابل الجريب فروت Grapefruit

انظر إلى هذه الفاكهة. هل هي برتقالة Orange أم جريب فروت Grapefruit؟ حسنًا، أعلم أن الجريب فروت بشكل عام أكبر وأكثر احمراراً.

Coding عربي

عملية تفكيري Thought Process هي شيء من هذا القبيل: لدي رسم بياني Graph في ذهني Mind.

بشكل عام، الفاكهة الأكبر والأكثر احمرارًا هي الجريب فروت Grapefruits. هذه الفاكهة كبيرة وحمراء اللون، لذا فهي على الأرجح ثمرة جريب فروت. لكن ماذا لو حصلت على فاكهة مثل هذه؟

كيف يمكنك تصنيف Classify هذه الفاكهة؟ إحدى الطرق هي النظر إلى جيران Neighbors هذا الموضع Spot. ألق نظرة على أقرب ثلاثة جيران Three Closest Neighbours لهذا الموضع.

البرتقال Oranges هم الجيران الأكثر من الجريب فروت Grapefruit. لذلك من المحتمل أن تكون هذه الفاكهة برتقالة Orange. تهانينا: لقد استخدمت للتو خوارزمية K-Nearest Neighbors Algorithm (KNN) للتصنيف Classification! الخوارزمية بأكملها بسيطة جدًا.

خوارزمية KNN بسيطة ولكنها مفيدة! إذا كنت تحاول تصنيف Classify شيء ما، فقد ترغب في تجربة خوارزمية KNN أولاً. دعونا نلقي نظرة على مثال أكثر واقعية Real-World Example.

بناء نظام اقتراحات
Recommendations System

لنفترض أنك شركة Netflix وتريد إنشاء نظام اقتراحات أفلام Movie Recommendations System من أجل المستخدمين Users. على مستوى عالٍ High Level، هذا مشابه لمسألة الجريب فروت Grapefruit Problem!

يمكنك تعيين موقع Plot كل مستخدم User على رسم بياني Graph.

يتم تعيين موقع Plot هؤلاء المستخدمين Users من خلال التشابه Similarity، لذلك يتم تعيين موقع Plotted المستخدمين ذوي الذوق المماثل Similar Taste بالقرب من بعضهم. افترض أنك تريد أن تقترح أفلام ل Priyanka. اعثر على أقرب خمسة مستخدمين Five Closest Users لها.

يتمتع كل من Justin و JC و Joey و Lance و Chris بذوق مماثل في الأفلام. لذا مهما كانت الأفلام التي يحبونها، فمن المحتمل Probably أن تعجب Priyanka أيضًا!

بمجرد الحصول على هذا الرسم البياني Graph، يصبح بناء نظام الاقتراحات - التوصيات Recommendations System أمرًا سهلاً. إذا كان Justin يحب فيلمًا، فاقترحه على Priyanka.

ولكن لا تزال هناك قطعة كبيرة مفقودة. لقد رسمت رسمًا بيانيًا Graphed للمستخدمين Users عن طريق التشابه Similarity. كيف يمكنك معرفة مدى تشابه اثنان من المستخدمين Two Users؟

استخراج الملامح Feature Extraction

في مثال الجريب فروت Grapefruit Example، قمت بمقارنة الفاكهة بناءً على حجمها ودرجة احمرارها. الحجم Size واللون Color هما الملمحان Features اللذان تقارنهما Compare. لنفترض الآن أن لديك ثلاث فواكه. يمكنك استخراج الملامح Extract Features.

ثم يمكنك عمل رسم بياني Graph للفواكه الثلاث.

من الرسم البياني Graph، يمكنك أن تقول بصريًا Visually أن الثمار A و B متشابهة Similar. دعونا نقيس مدى قربهم Near - Close. لإيجاد المسافة بين نقطتين، استخدم معادلة فيثاغورس Pythagorean Formula.

$$\sqrt{(X_1-X_2)^2+(Y_1-Y_2)^2}$$

هذه هي المسافة بين A و B، على سبيل المثال.

$$\sqrt{(2-2)^2 + (2-1)^2}$$
$$= \sqrt{0+1}$$
$$= \sqrt{1}$$
$$= 1$$

المسافة بين A و B هي 1. يمكنك أن تجد باقي المسافات Distances أيضًا.

تؤكد معادلة المسافة Distance Formula ما رأيته بصريًا Visually: الفاكهة A و B متشابهة Similar.

بدلاً من ذلك لنفترض أنك تقارن مستخدمي Netflix. أنت بحاجة إلى طريقة ما للرسم البياني للمستخدمين Graph The Users. لذلك، تحتاج إلى تحويل Convert كل مستخدم User إلى مجموعة من الإحداثيات Set of Coordinates، تمامًا كما فعلت مع الفاكهة.

بمجرد أن تتمكن من الرسم البياني للمستخدمين Graph Users، يمكنك قياس المسافة Distance بينهم.

إليك كيفية تحويل المستخدمين إلى مجموعة من الأرقام Set Of Numbers. عندما يشترك Sign Up المستخدمون في Netflix، اطلب منهم تقييم Rate بعض فئات الأفلام Movies Categories بناءً على مدى إعجابهم بهذه الفئات. لكل مستخدم، لديك الآن مجموعة من التقييمات Set Of Ratings!

		PRIYANKA	JUSTIN	MORPHEUS
كوميدي	COMEDY	3	4	2
أكشن	ACTION	4	3	5
دراما	DRAMA	4	5	1
رعب	HORROR	1	1	3
رومانسي	ROMANCE	4	5	1

بريانكا Priyanka وجاستين Justin يحبون أفلام الرومانسية Romance ويكرهون أفلام الرعب Horror. يحب Morpheus أفلام الأكشن Action ولكنه يكره أفلام الرومانسية Romance (يكره عندما يُفسد مشهد رومانسي فيلم أكشن Action جيد). تذكر في مسألة البرتقال Oranges في مقابل الجريب فروت Grapefruit كيف تم تمثيل Represented كل فاكهة بمجموعة من رقمين Set Of Two Numbers؟ هنا، يتم تمثيل كل مستخدم User بمجموعة من خمسة أرقام.

قد يقول عالم رياضيات Mathematician، بدلاً من حساب المسافة في بُعدِين Two Dimensions، أنت الآن تحسب المسافة في خمسة أبعاد Five Dimensions. لكن معادلة المسافة Distance Formula تظل كما هي.

$$\sqrt{(a_1-a_2)^2+(b_1-b_2)^2+(c_1-c_2)^2+(d_1-d_2)^2+(e_1-e_2)^2}$$

إنها تتضمن فقط مجموعة Set من خمسة أرقام بدلاً من مجموعة من رقمين.

معادلة المسافة Distance Formula مرنة Flexible: يمكن أن يكون لديك مجموعة من مليون رقم وما زلت تستخدم نفس معادلة المسافة القديمة لإيجاد المسافة. ربما تتساءل، "ماذا تعني المسافة Distance عندما يكون لديك خمسة أرقام؟" تخبرك المسافة عن مدى تشابه Similarity مجموعات الأرقام هذه.

$$\sqrt{(3-4)^2+(4-3)^2+(4-5)^2+(1-1)^2+(4-5)^2}$$
$$=\sqrt{1+1+1+0+1}$$
$$=\sqrt{4}$$
$$=2$$

ها هي المسافة بين Priyanka و Justin.

Priyanka و Justin متشابهان إلى حد كبير. ما الفرق بين Priyanka و Morpheus؟ احسب المسافة Distance قبل مواصلة القراءة.

هل فهمتها جيدا؟ Priyanka و Morpheus يفصل بينهما مسافة 24. تخبرك المسافة أن ذوق Priyanka يشبه ذوق Justin أكثر من ذوق Morpheus.

رائع! الآن أصبح اقتراح أفلام Recommending Movies على Priyanka أمرًا سهلاً: إذا أحب Justin فيلمًا، إذاً قم باقتراحه على Priyanka، والعكس صحيح. لقد بنيت للتو نظام اقتراحات أفلام Movie Recommendations System!

إذا كنت أحد مستخدمي Netflix، فستستمر Netflix في إخبارك، "يرجى تقييم Rate المزيد من الأفلام. كلما زاد عدد الأفلام التي تقوم بتقييمها، ستكون الاقتراحات Recommendations المقدمة لك أفضل ". الآن أنت تعرف لماذا. كلما زاد عدد الأفلام التي تقوم بتقييمها، زادت دقة رؤية Netflix لمدي تشابهك Similar مع مستخدمون آخرون.

التمارين Exercises

10.1 في مثال Netflix، قمت بحساب المسافة Distance بين مختلف المستخدمين Different Users باستخدام معادلة المسافة Distance Formula. لكن لا يقوم كل المستخدمين Users بتقييم Rate الأفلام بنفس الطريقة. لنفترض أن لديك اثنان من المستخدمين، Yogi و Pinky، لهما نفس الأذواق في الأفلام. لكن Yogi يقوم بتقييم أي فيلم يحبه بتقييم 5، في حين أن Pinky أكثر انتقائية ويقوم بحجز تقييم 5 للأفضل فقط. إنهما متطابقان بشكل جيد، لكن وفقًا لخوارزمية المسافة Distance Algorithm، فهما ليسا جيران Neighbors. كيف ستأخذ استراتيجياتهم المختلفة للتقييم Different Rating Strategies في الاعتبار؟

10.2 لنفترض أن Netflix تختار مجموعة من "المؤثرين Influencers". على سبيل المثال، يعتبر Quentin Tarantino و Wes Anderson من المؤثرين على Netflix، لذا فإن تقييماتهم Ratings يجب تقديرها أكثر من تقييمات المستخدمين العاديين. كيف يمكنك تغيير نظام الاقتراحات Recommendations System بحيث يكون منحازًا Biased لتقييمات المؤثرين؟

الانحدار Regression

لنفترض أنك تريد أن تفعل أكثر من مجرد اقتراح فيلم: أنت تريد أن تُخمِّن Guess كيف ستقوم Priyanka بتقييم Rate هذا الفيلم. خذ أقرب Nearest خمسة أشخاص إليها.

بالمناسبة، أتحدث باستمرار عن أقرب خمسة أشخاص. لا يوجد شيء مميز في الرقم 5: يمكنك عمل أقرب 2 أو 10 أو 10000. هذا هو السبب في أن الخوارزمية تسمى أقرب عدد K من الجيران K-Nearest Neighbors وليس أقرب خمسة جيران 5-Nearest Neighbors!

لنفترض أنك تحاول تخمين تقييم فيلم Pitch Perfect. حسنًا، كيف قام Justin و JC و Joey و Lance و Chris بتقييمه؟

يمكنك الحصول على متوسط Average تقييماتهم Ratings والحصول على 4.2 نجوم. هذا يسمى الانحدار - الانحسار Regression. هذان هما الشيئان الأساسيان اللذان ستفعلهما بواسطة KNN – التصنيف Classification والانحدار Regression:

- التصنيف Classification = التصنيف Categorization إلي مجموعة Group
- الانحدار Regression = توقع استجابة Predicting Response (مثل توقع رقم Predicting Number)

الانحدار Regression مفيد جدا. لنفترض أنك تدير مخبزًا صغيرًا في بيركلي، وأنك تصنع خبزًا طازجًا كل يوم. إنك تحاول توقع Predict عدد الأرغفة التي ستصنعها لهذا اليوم. لديك مجموعة من الملامح Features:

- الطقس Weather على مقياس من 1 إلى 5 (1 = سيء Bad، 5 = عظيم Great).
- عطلة نهاية الأسبوع Weekend أو العطلة Holiday؟ (1 إذا كانت Weekend أو Holiday ، 0 إذا كانت غير ذلك)
- هل هناك مباراة؟ (1 إذا كانت الإجابة بنعم، 0 إذا كانت الإجابة لا)

وأنت تعرف عدد أرغفة الخبز Loaves التي بعتها في الماضي لمجموعات مختلفة من الملامح Features.

اليوم هو يوم عطلة نهاية الأسبوع Weekend مع طقس جيد Good Weather. بناءً على البيانات Data التي رأيتها للتو، كم عدد الأرغفة التي ستبيعها؟ دعنا نستخدم خوارزمية KNN، حيث K=4. أولاً، اكتشف أقرب أربع جيران Four Nearest Neighbors لهذه النقطة Point.

فيما يلي المسافات. A و B و D و E هي الأقرب Closest.

خذ متوسط Average عدد الأرغفة التي تم بيعها في تلك الأيام، وستحصل على 218.75. هذا هو عدد الأرغفة التي يجب أن تصنعها لهذا اليوم!

تشابه جيب التمام Cosine Similarity

حتى الآن، كنت تستخدم معادلة المسافة Distance Formula لمقارنة المسافة بين اثنين من المستخدمين Two Users. هل هذه هي أفضل معادلة للاستخدام؟ أحد المعادلات شائعة الاستخدام في الممارسة هي تشابه جيب التمام Cosine Similarity. لنفترض أن هناك مستخدمين اثنين متشابهين Similar Two Users، لكن أحدهما أكثر تحفظًا Conservative في تقييماته Ratings. كلاهما أحب فيلم Amar Akbar Anthony ل Manmohan Desai. قام بول Paul بتقييمه بـ 5 نجوم، لكن روان Rowan قام بتقييمه بـ 4 نجوم. إذا واصلت استخدام معادلة المسافة Distance Formula، فقد لا يكون هذان المستخدمان Two Users جيران Neighbors لبعضهما البعض، على الرغم من ذوقهما المتشابه Similar Taste.

تشابه جيب التمام Cosine Similarity لا يقيس Measure المسافة بين متجهين Two Vectors. بدلاً من ذلك، فإنه يُقارن زوايا Angle المتجهين Two Vectors. إنه أفضل في التعامل مع مثل هذه الحالات Cases. تشابه جيب التمام Cosine Similarity خارج نطاق هذا الكتاب، لكن ابحث عنها إذا كنت تستخدم KNN!

انتقاء الملامح الجيدة Picking Good Features

لاكتشاف الاقتراحات Recommendations، فقد طلبت من المستخدمين تقييم فئات Categories الأفلام. ماذا لو جعلتهم يقومون بتقييم صور القطط بدلاً من ذلك؟ ثم ستجد المستخدمين الذين قاموا بتقييم تلك الصور بشكل مُشابه Similarly. قد يكون هذا مُحرِّك اقتراحات أسوأ Worse Recommendations Engine، لأن الملامح Features ليس لها علاقة كبيرة بالذوق في الأفلام!

أو افترض أنك تطلب من المستخدمين تقييم الأفلام حتى تتمكن من إعطائهم اقتراحات — لكنك تطلب منهم فقط تقييم Toy Story و Toy Story 2 و Toy Story 3. لن يخبرك هذا كثيرًا عن أذواق المستخدمين للأفلام!

عندما تعمل مع KNN، من المهم حقًا اختيار الملامح Features الصحيحة للمقارنة من خلالها. اختيار الملامح Features الصحيحة يعني

- الملامح Features التي ترتبط ارتباطًا مباشرًا Directly Correlate بالأفلام التي تحاول التوصية بها – اقتراحها Recommend
- الملامح Features التي لا تحتوي على تَحيُّز Bias (على سبيل المثال، إذا طلبت من المستخدمين تقييم الأفلام الكوميدية فقط، فهذا لا يخبرك ما إذا كانوا يحبون أفلام الأكشن Action)

هل تعتقد أن التقييمات Ratings طريقة جيدة لاقتراح Recommend أفلام؟ ربما أنا قمت بتقييم فيلم The Wire بدرجة أعلى من House Hunters، لكنني في الواقع أقضي وقتًا أطول في مشاهدة House Hunters.

كيف يمكنك تحسين نظام توصيات - اقتراحات Netflix هذا؟

بالعودة إلى المخبز: هل يمكنك التفكير في ملمحين Two Features جيدين وملمحين سيئين كان من الممكن أن تختارهما للمخبز؟ ربما تحتاج إلى عمل المزيد من الأرغفة بعد الإعلان في الجريدة. أو ربما تحتاج إلى صنع المزيد من الأرغفة أيام الاثنين.

لا توجد إجابة واحدة صحيحة عندما يتعلق الأمر باختيار الملامح الجيدة Good Features. عليك أن تفكر في كل الأشياء المختلفة التي تحتاج إلى أخذها في الاعتبار.

التمرين Exercise

10.3 لدى Netflix ملايين المستخدمين. نظر المثال السابق في أقرب خمسة جيران Five Closest Neighbors لبناء نظام التوصيات – الاقتراحات Recommendations System. هل هذا منخفض جدا Too Low؟ أو مرتفع جدا To High؟

مقدمة في التعلم الآلي
Introduction To Machine Learning

KNN هي خوارزمية Algorithm مفيدة حقًا، وهي مقدمتك إلى العالم السحري للتعلم الآلي Machine Learning! يدور التعلم الآلي حول جعل جهاز الكمبيوتر الخاص بك أكثر ذكاءً. لقد رأيت بالفعل أحد الأمثلة على

التعلم الآلي Machine Learning: بناء نظام توصيات – اقتراحات Recommendations System. دعونا نلقي نظرة على بعض الأمثلة الأخرى.

التعرف البصري على الحروف OCR

يرمز OCR إلى التعرف البصري على الحروف Optical Character Recognition. هذا يعني أنه يمكنك التقاط صورة Photo لصفحة نصية Page Of Text، وسيقوم جهاز الكمبيوتر الخاص بك بقراءة Read النص Text تلقائيًا Automatically نيابة عنك. تستخدم Google التعرف البصري على الحروف OCR لرقمنة الكتب Digitize Books. كيف يعمل OCR؟ على سبيل المثال، انظر إلى هذا الرقم.

كيف ستكتشف تلقائيًا ما هو هذا الرقم؟ يمكنك استخدام KNN لهذا:

1. تصفح الكثير من صور Images الأرقام Numbers وقم باستخراج Extract ملامح Features تلك الأرقام.
2. عندما تحصل على صورة جديدة New Image، قم باستخراج ملامح Features تلك الصورة، وانظر ما هي أقرب جيرانها Nearest Neighbors!

إنها نفس مسألة البرتقال Oranges مقابل الجريب فروت Grapefruit. بشكل عام، خوارزميات OCR تقيس Measure الخطوط Lines والنقاط Points والمنحنيات Curves.

بعد ذلك، عندما تحصل على حرف جديد New Character، يمكنك استخراج نفس الملامح Features منها. يعد استخراج الملامح Feature Extraction أكثر تعقيدًا في التعرف البصري على الحروف OCR من مثال الفاكهة. ولكن من المهم أن نفهم أنه حتى التقنيات المعقدة Complex Technologies تبني على أفكار بسيطة، مثل KNN. يمكنك استخدام نفس الأفكار للتعرف على الكلام Speech Recognition أو التعرف على الوجوه

Face Recognition. عند رفع Upload صورة على Facebook، يكون فيسبوك من الذكاء أحيانًا بحيث يضع وسم Tag على الأشخاص الموجودين في الصورة تلقائيًا. هذا هو التعلم الآلي Machine Learning أثناء العمل In Action!

الخطوة الأولى في التعرف البصري على الحروف OCR، حيث تقوم بتصفح صور Images الأرقام Numbers واستخراج الملامح Extract Features، تسمى بالتدريب Training. معظم خوارزميات التعلم الآلي Machine-Learning Algorithms لديها خطوة تدريب Training Step: قبل أن يتمكن جهاز الكمبيوتر الخاص بك من القيام بالمهمة، يجب تدريبه. يتضمن المثال التالي فلاتر - مُرشِّحَات البريد العشوائي - المزعج Spam Filters، ولديه خطوة تدريب Training Step.

بناء فلتر البريد العشوائي Spam Filter

تستخدم فلاتر - مُرَشِّحات البريد المزعج Spam Filters خوارزمية بسيطة أخرى تسمى مُصنِّف نايف بايز Naive Bayes Classifier. أولاً، تقوم بتدريب مصنف Naive Bayes Classifier الخاص بك على بعض البيانات Data.

لنفترض أنك تلقيت بريدًا إلكترونيًا Email بعنوان "اجمع ملايين الدولارات الآن!" هل هو بريد مزعج Spam؟ يمكنك تقسيم هذه الجملة Sentence إلى كلمات Words. ثم، لكل كلمة Word، انظر إلى احتمال Probability ظهور هذه الكلمة في بريد إلكتروني عشوائي - مزعج Spam Email. على سبيل المثال، في هذا النموذج Model البسيط للغاية، تظهر كلمة مليون Million فقط في رسائل البريد الإلكتروني العشوائية Spam Emails.

تكتشف خوارزمية Naive Bayes احتمال Probability أن يكون هناك شيء ما من المحتمل أن يكون بريدًا عشوائيًا Spam. لديها تطبيقات مشابهة لـ KNN.

على سبيل المثال، يمكنك استخدام Naive Bayes لتصنيف Categorize الفاكهة: لديك فاكهة كبيرة وحمراء. ما هو احتمال أن تكون جريب فروت Grapefruit؟ إنها خوارزمية بسيطة أخرى فعالة Effective بشكل مقبول. نحن نحب تلك الخوارزميات Algorithms!

توقع سوق الأسهم
Predicting Stock Market

إليك أمر يصعب القيام به بواسطة التعلم الآلي Machine Learning: التنبؤ Predicting حقًا بما إذا كان سوق الأسهم Stock Market سيرتفع أم ينخفض. كيف تختار الملامح Features الجيدة في سوق الأوراق المالية؟ لنفترض أنك قلت إنه إذا ارتفع السهم Stock أمس، فسوف يرتفع اليوم. هل هذا ملمح Feature جيد؟ أو افترض أنك تقول إن السهم سينخفض دائمًا في شهر مايو. هل سينجح هذا؟ لا توجد طريقة مضمونة لاستخدام الأرقام السابقة لتوقع الأداء المستقبلي. من الصعب التنبؤ بالمستقبل، ويكاد يكون من المستحيل عندما يكون هناك الكثير من المتغيرات Variables المتضمنة.

الخلاصة

آمل أن يمنحك هذا فكرة عن جميع الأشياء المختلفة التي يمكنك القيام بها باستخدام KNN والتعلم الآلي Machine Learning! يعد التعلم الآلي مجالًا مثيرًا للاهتمام يمكنك التعمق فيه إذا قررت ذلك:

- يُستخدم KNN في التصنيف Classification والانحدار Regression ويشمل النظر إلى أقرب عدد K من الجيران K-Nearest Neighbors.
- التصنيف Classification = التصنيف Categorization في مجموعة Group
- الانحدار Regression = توقع استجابة Predicting Response (مثل توقع رقم Number)
- استخراج الملامح Feature Extraction يعني تحويل عنصر Converting Item (مثل فاكهة أو مستخدم User) إلى قائمة أرقام List of Numbers يمكن مقارنتها.
- يعد اختيار الملامح Features الجيدة جزءًا مهمًا من خوارزمية KNN ناجحة.

11 إلى أين نذهب بعد ذلك
Where To Go Next

في هذا الفصل

- يمكنك الحصول على نظرة عامة مختصرة على 10 خوارزميات Algorithms لم يتم تناولها في هذا الكتاب ولماذا هي مفيدة.
- تحصل على مؤشرات حول ما ستقرأه بعد ذلك، اعتمادًا على اهتماماتك.

الأشجار Trees

دعنا نعود إلى مثال البحث الثنائي Binary Search. عندما يقوم المستخدم بتسجيل الدخول إلى Facebook، يتعين على Facebook البحث في مصفوفة Array كبيرة لمعرفة ما إذا كان اسم المستخدم Username موجودًا. قلنا إن أسرع طريقة للبحث من خلال هذه المصفوفة هي إجراء بحث ثنائي Binary Search. ولكن هناك مشكلة: في كل مرة يقوم فيها مستخدم جديد New User بالتسجيل Sign Up، تقوم بإدراج Insert اسم المستخدم الخاص به في المصفوفة Array. ثم يتعين عليك إعادة ترتيب Re-Sort المصفوفة، لأن البحث الثنائي Binary Search يعمل فقط مع المصفوفات التي تم ترتيبها Sorted. ألن يكون من الجيد إدراج Insert اسم المستخدم في الخانة اليمنى Right Slot في المصفوفة على الفور، حتى لا تضطر إلى ترتيب المصفوفة بعد ذلك؟ هذه هي الفكرة من وراء هيكل بيانات شجرة البحث الثنائي Binary Search Tree Data Structure.

تبدو شجرة البحث الثنائي Binary Search Tree هكذا.

لكل عقدة Node، العقد Nodes الموجودة على يسارها تكون أصغر في القيمة Value، والعقد على يمينها أكبر في القيمة.

لنفترض أنك تبحث عن Maggie. ستبدأ من عقدة الأصل Root Node.

تأتي Maggie بعد David، لذا اتجه نحو اليمين Right.

تأتي Maggie قبل Manning، لذا اذهب إلى اليسار Left.

لقد وجدت Maggie! يكاد يكون مثل إجراء بحث ثنائي Binary Search! يستغرق البحث عن عنصر Element في شجرة البحث الثنائي Binary Search Tree وقت $O(\log n)$ في المتوسط Average و $O(n)$ في أسوأ حالة Worst Case. يستغرق البحث في مصفوفة مرتبة Sorted Array وقت $O(\log n)$ في أسوأ حالة Worst Case، لذلك قد تعتقد أن المصفوفة المرتبة Sorted Array أفضل. لكن شجرة البحث الثنائي Binary Search Tree أسرع كثيرًا لعمليات الإدراج – الإدخال Insertions و عمليات الحذف Deletions في المتوسط Average.

	المصفوفة ARRAY	شجرة البحث الثنائي BINARY SEARCH TREE
البحث SEARCH	$O(\log n)$	$O(\log n)$
الإدراج INSERT	$O(n)$	$O(\log n)$
الحذف DELETE	$O(n)$	$O(\log n)$

تحتوي أشجار البحث الثنائي Binary Search Trees على بعض الجوانب السلبية أيضًا: لسبب واحد، لا تحصل على وصول عشوائي Random Access. لا يمكنك أن تقول، "أعطني العنصر الخامس من هذه الشجرة Tree."
أوقات الأداء Performance Times هذه أيضًا في المتوسط Average وتعتمد على عمل توازن Balance للشجرة Tree. افترض أن لديك شجرة غير متوازنة Imbalanced Tree مثل تلك التي تظهر في الصورة التالية.

انظر كيف تميل Lean الشجرة إلى اليمين؟ لا تتمتع هذه الشجرة بأداء Performance جيد جدًا، لأنها غير متوازنة. هناك أشجار بحث ثنائي خاصة Special Binary Search Trees تُوازن Balance نفسها. أحد الأمثلة على ذلك هو الشجرة ذات اللون الأحمر والأسود Red-Black Tree.
إذن متى يتم استخدام أشجار البحث الثنائي Binary Search Trees؟ أشجار B-Trees، وهي نوع خاص Special Type من الأشجار الثنائية Binary Tree، تُستخدم بشكل شائع لتخزين البيانات Store Data في قواعد البيانات Databases.
إذا كنت مهتمًا بقواعد البيانات Databases أو هياكل بيانات أكثر تقدمًا More Advanced Data Structures، فابحث عن هذه:

- B-Trees
- أشجار حمراء-سوداء Red-Black Trees
- المكدسات Heaps
- أشجار سبلاي Splay Trees

الفهارس المعكوسة Inverted Indexes

فيما يلي نسخة مبسطة جدًا عن كيفية عمل محرك البحث Search Engine. لنفترض أن لديك ثلاث صفحات ويب Web Pages بها هذا المحتوى Content البسيط.

دعونا نبني جدول تجزئة Hash Table من هذا المحتوى Content.

مفاتيح Keys جدول التجزئة هي الكلمات Words، والقيم Values تخبرك بالصفحات Pages التي تظهر عليها كل كلمة Word. افترض الآن أن مستخدمًا User يبحث عن *hi*. دعونا نرى ما هي الصفحات التي تظهر فيها كلمة *hi*.

حسناً: تظهر في الصفحات A و B. فلنعرض هذه الصفحات Pages للمستخدم كنتيجة Result. أو افترض أن المستخدم يبحث عن *there*. حسنًا، أنت تعلم أنها تظهر على الصفحات A و C، هذا سهل جدًا، أليس كذلك؟ هذا هيكل بيانات Data Structure مفيد: تجزئة Hash تقوم بتعيين Map الكلمات Words إلى الأماكن التي تظهر فيها. يسمى هيكل البيانات هذا بالفهرس المعكوس- المقلوب Inverted Index، وهي تستخدم بشكل شائع لبناء محركات البحث Search Engines. إذا كنت مهتمًا بالبحث، فهذا مكان جيد للبدء.

تحويل فورييه The Fourier Transform

يعد تحويل فورييه Fourier Transform أحد تلك الخوارزميات Algorithms النادرة: رائعة وأنيقة Elegant ولها مليون حالة استخدام Use Cases. أفضل تشبيه لتحويل فورييه يأتي من موقع الويب Better Explained (موقع ويب رائع يشرح الرياضيات Math ببساطة): عند إعطاؤه عصير، سيخبرك تحويل فورييه

Fourier Transform بالمكونات الموجودة في العصير. أو بعبارة أخرى، عند إعطاؤه أغنية، يمكن أن يفصلها التحويل Fourier Transform إلى ترددات فردية Individual Frequencies.

اتضح أن هذه الفكرة البسيطة لها الكثير من حالات الاستخدام Use Cases. على سبيل المثال، إذا كان بإمكانك فصل أغنية إلى ترددات Frequencies، فيمكنك تعزيز Boost الأغاني التي تهتم بها. يمكنك تعزيز الصوت الجهير Bass وإخفاء الطبقة الثلاثية Treble. يعتبر تحويل فورييه Fourier Transform رائعًا لمعالجة الإشارات Processing Signals. يمكنك أيضًا استخدامه لضغط الموسيقى Compress Music. أولاً تقوم بتقسيم ملف صوتي Audio File إلى مكوناته من النوتات الموسيقية Notes. يخبرك تحويل فورييه Fourier Transform بالضبط كم تساهم كل نغمة - نوتة في الأغنية ككل. لذلك يمكنك التخلص من النوتات - النغمات غير المهمة. هذه هي الطريقة التي تعمل بها صيغة MP3 Format!

الموسيقى ليست النوع الوحيد من الإشارات الرقمية Digital Signal. صيغة JPG Format هي صيغة مضغوطة Compressed Format أخري، وتعمل بنفس الطريقة. يستخدم الناس تحويل فورييه Fourier Transform لمحاولة التنبؤ بالزلازل القادمة وتحليل الحمض النووي Analyze DNA.

يمكنك استخدامه لإنشاء تطبيق مثل Shazam، والذي يخمن الأغنية التي يتم تشغيلها. لتحويل فورييه الكثير من الاستخدامات. من المحتمل بشكل كبير أن تقابلها!

الخوارزميات المتوازية Parallel Algorithms

تدور الموضوعات الثلاثة التالية حول قابلية التوسع Scalability والعمل مع الكثير من البيانات Data. في الماضي، أخذت أجهزة الكمبيوتر تزداد سرعةً بشكل تدريجي. إذا كنت ترغب حينها في جعل الخوارزمية Algorithm الخاصة بك أسرع، كان يمكنك الانتظار بضعة أشهر، وستصبح أجهزة الكمبيوتر نفسها أسرع. لكننا اقتربنا من نهاية تلك الحقبة. بدلاً من ذلك، تأتي أجهزة الكمبيوتر المحمولة Laptops وأجهزة الكمبيوتر Computers مع أنوية متعددة Multiple Cores. لجعل الخوارزميات Algorithms الخاصة بك أسرع، تحتاج إلى تغييرها لتعمل بالتوازي Run In Parallel عبر جميع الأنوية Cores في وقت واحد!

فيما يلي مثال بسيط. أفضل ما يمكنك فعله باستخدام خوارزمية ترتيب Sorting Algorithm هو تقريبًا وقت $O(n \log n)$. من المعروف أنه لا يمكنك ترتيب مصفوفة Sort Array في وقت $O(n)$ - إلا إذا كنت تستخدم خوارزمية متوازية Parallel Algorithm! هناك نسخة متوازية Parallel Version من الترتيب السريع Quicksort والتي سوف تقوم بترتيب مصفوفة في وقت $O(n)$.

الخوارزميات المتوازية Parallel Algorithms صعبة التصميم Design. ومن الصعب أيضًا التأكد من أنها تعمل بشكل صحيح ومعرفة نوع زيادة السرعة Speed Boost التي ستراها. هناك شيء واحد مؤكد - المكاسب الزمنية ليست خطية Linear. لذا، إذا كان لديك اثنان من الأنوية Cores في الكمبيوتر المحمول Laptop بدلاً

من واحد، فهذا تقريبا لا يعني أبدًا أن الخوارزمية Algorithm الخاصة بك ستعمل Run بشكل سحري بسرعة مضاعفة. هناك عدة أسباب لذلك:

- التكاليف غير المباشرة Overhead لإدارة التوازي Managing Parallelism - لنفترض أن عليك ترتيب مصفوفة Sort Array من 1000 عنصر. كيف تقسم هذه المهمة بين اثنان من الأنوية Two Cores؟ هل تعطي كل نواة 500 عنصر لترتيبهم ثم دمج Merge المصفوفتين المرتبتين Two Sorted Arrays في مصفوفة واحدة كبيرة مرتبة؟ يستغرق دمج المصفوفتين وقتًا.
- موازنة الحمل Load Balancing - لنفترض أن لديك 10 مهام للقيام بها، لذا فإنك تعطي كل نواة 5 مهام. لكن النواة A تحصل على جميع المهام السهلة، لذلك يتم إنجازها في 10 ثوانٍ، بينما تتولى النواة B جميع المهام الصعبة، لذلك يستغرق الأمر دقيقة. هذا يعني أن النواة A كانت جالسة في خاملة Idle لمدة 50 ثانية بينما كانت النواة B تقوم بكل العمل! كيف تُوزِّع العمل بالتساوي بحيث يعمل كلا النواتان Both Cores بجهد متساوٍ؟

إذا كنت مهتمًا بالجانب النظري للأداء Performance وقابلية التوسع Scalability، فقد تكون الخوارزميات المتوازية Parallel Algorithms مناسبة لك!

MapReduce

هناك نوع خاص من الخوارزمية المتوازية Parallel Algorithm يزداد شيوعًا: الخوارزمية الموزَّعة Distributed Algorithm. من الجيد تشغيل Run خوارزمية متوازية Parallel Algorithm على الكمبيوتر المحمول Laptop إذا كنت تحتاج إلى اثنين إلى أربعة أنوية Cores، ولكن ماذا لو كنت بحاجة إلى مئات الأنوية Cores؟ ومن ثم يمكنك كتابة الخوارزمية Algorithm الخاصة بك للتشغيل Run عبر Across أجهزة متعددة Multiple Machines. خوارزمية MapReduce هي خوارزمية موزَّعة شائعة Popular Distributed Algorithm. يمكنك استخدامها من خلال الأداة مفتوحة المصدر Open Source Tool وهي Apache Hadoop.

لماذا الخوارزميات الموزَّعة Distributed Algorithms مفيدة؟

لنفترض أن لديك جدولاً Table به بلايين أو تريليونات من الصفوف Rows، وتريد تشغيل Run استعلام سيكويل معقد Complicated SQL Query عليه. لا يمكنك تشغيله على MySQL، لأنها تتقدم بصعوبة بعد بضع مليارات من الصفوف Rows. استخدم MapReduce من خلال Hadoop!

أو افترض أن عليك معالجة Process قائمة طويلة من المهام Jobs. تستغرق كل مهمة 10 ثوانٍ للمعالجة، وتحتاج إلى معالجة مليون مهمة كهذه. إذا قمت بذلك على جهاز واحد، فسوف يستغرق الأمر شهورًا! إذا كان بإمكانك تشغيلها عبر 100 جهاز، فقد تنتهي في غضون أيام قليلة.

الخوارزميات الموزعة Distributed Algorithms رائعة عندما يكون لديك الكثير من العمل الذي يتعين عليك القيام به وترغب في تسريع الوقت المطلوب للقيام بذلك. تم إنشاء MapReduce على وجه الخصوص من فكرتين بسيطتين: الدالة map والدالة reduce.

الدالة Map

الدالة map بسيطة: فهي تأخذ Take مصفوفة Array وتُطبِّق نفس الدالة Function على كل عنصر Item في المصفوفة. على سبيل المثال، نقوم هنا بمضاعفة Doubling كل عنصر في المصفوفة:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = map(lambda x: 2 * x, arr1)
[2, 4, 6, 8, 10]
```

تحتوي arr2 الآن على [2, 4, 6, 8, 10] —تم مضاعفة كل عنصر في arr1! مضاعفة العنصر سريعة جدًا. لكن لنفترض أنك قمت بتطبيق دالة Apply Function تستغرق وقتًا أطول في المعالجة Processing.

انظر إلى هذا الكود الزائف Pseudocode:

```
>>> arr1 = # A list of URLs
>>> arr2 = map(download_page, arr1)
```

هنا لديك قائمة بعناوين URL، وتريد تنزيل Download كل صفحة Page وتخزين المحتويات Store Contents في arr2. قد يستغرق هذا بضع ثوانٍ لكل عنوان URL. إذا كان لديك 1000 عنوان URL، فقد يستغرق ذلك بضع ساعات!

ألن يكون رائعًا لو كان لديك 100 جهاز، ويمكن لدالة map أن تنشر العمل تلقائيًا عبر كل منهم؟ إذًا ستقوم بتنزيل 100 صفحة في المرة الواحدة، وسيمضي العمل أسرع كثيرًا! هذه هي الفكرة من وراء "Map" في MapReduce.

الدالة Reduce

الدالة reduce تقوم بإرباك الناس في بعض الأحيان. الفكرة هي أن تقوم "باختزال – تقليص - تخفيض" قائمة كاملة من العناصر إلى عنصر واحد. باستخدام map، يمكنك الانتقال من مصفوفة Array إلى أخرى.

باستخدام reduce، يمكنك تحويل Transform مصفوفة إلى عنصر واحد.

هذا مثال:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> reduce(lambda x,y: x+y, arr1)
15
```

في هذه الحالة، تقوم بجمع Sum Up كل العناصر في المصفوفة: 1 + 2 + 3 + 4 + 5 = 15!

لن أشرح reduce بمزيد من التفصيل هنا، لأن هناك الكثير من الدروس التعليمية Tutorials عبر الإنترنت. تستخدم خوارزمية MapReduce هذين المفهومين البسيطين لتشغيل Run استعلامات Queries حول البيانات Data عبر أجهزة متعددة Multiple Machines. عندما يكون لديك مجموعة بيانات Dataset كبيرة (بلايين الصفوف Billions Of Rows)، يمكن أن تمنحك MapReduce إجابة في دقائق بينما قد تستغرق قاعدة البيانات التقليدية Traditional Database ساعات.

فلاتر - مُرشِّحات بلوم Bloom Filters و HyperLogLog

لنفترض أنك تُدير موقع Reddit. عندما ينشر Post شخص ما رابط Link، فأنت تريد معرفة ما إذا كان قد تم نشره من قبل. تعتبر القصص Stories التي لم يتم نشرها من قبل أكثر قيمة. لذلك تحتاج إلى معرفة ما إذا كان هذا الرابط قد تم نشره من قبل.

أو افترض أنك محرك البحث Google، وأنت تزحف Crawling خلال صفحات الويب Web Pages. أنت تريد فقط الزحف Crawl خلال صفحة ويب لم تقم بالزحف خلالها من قبل. لذلك تحتاج إلى معرفة ما إذا كان قد تم الزحف Crawled خلال هذه الصفحة من قبل.

أو افترض أنك تقوم بإدارة موقع bit.ly، وهو أداة تقصير Shortener لعناوين URL. لا تريد إعادة توجيه Redirect المستخدمين Users إلى مواقع الويب الضارة Malicious Websites. لديك مجموعة Set من عناوين URL التي تعتبر ضارة Malicious. أنت الآن بحاجة إلى معرفة ما إذا كنت تعيد توجيه Redirecting المستخدم إلى عنوان URL في تلك المجموعة.

كل هذه الأمثلة لها نفس المشكلة. لديك مجموعة Set كبيرة جدا.

الآن لديك عنصر جديد New Item، وتريد معرفة ما إذا كان ينتمي إلى هذه المجموعة Set. يمكنك القيام بذلك بسرعة باستخدام التجزئة Hash. على سبيل المثال، لنفترض أن Google لديه تجزئة Hash كبيرة حيث تكون المفاتيح Keys هي جميع الصفحات Pages التي قام بالزحف Crawled خلالها.

تريد معرفة ما إذا كنت قد قمت بالفعل بالزحف Crawled خلال موقع instagram.com. ابحث عنها في التجزئة Hash.

وجدت instagram.com كمفتاح Key في التجزئة Hash، لذا فقد قمت بالزحف Crawled خلاله بالفعل.

متوسط وقت البحث Average Lookup Time لجداول التجزئة هو $O(1)$. لقد اكتشفت ذلك في وقت ثابت Constant Time. جيد جدا!

باستثناء أن هذا الهاش Hash يجب أن يكون ضخمًا. تقوم Google بفهرسة Indexing تريليونات من صفحات الويب Web Pages. إذا كانت هذه التجزئة Hash تحتوي على جميع عناوين URL التي فهرستها Google، فستشغل مساحة Space كبيرة. لدى كل من موقعي Reddit و bit.ly نفس مشكلة المساحة. عندما يكون لديك الكثير من البيانات Data، عليك أن تكون مُبدعًا Creative!

فلاتر بلوم Bloom Filters

تقدم فلاتر بلوم Bloom Filters حلاً Solution. مرشحات بلوم Bloom هي هياكل بيانات احتمالية Probabilistic Data Structures. يعطونك إجابة قد تكون خاطئة من المحتمل أن تكون صحيحة. بدلاً من التجزئة Hash، يمكنك سؤال فلتر بلوم Bloom Filter عما إذا كنت قد زحفت Crawled خلال العنوان URL هذا من قبل. يمنحك جدول التجزئة Hash Table إجابة دقيقة. سيمنحك فلتر بلوم Bloom Filter إجابة من المحتمل أن تكون صحيحة:

- التأكيدات الخاطئة False Positives محتملة - ممكنة Possible. قد يقول محرك البحث Google، "لقد قمت بالفعل بالزحف Crawled خلال هذا الموقع"، على الرغم من أنك لم تقم بذلك.
- النفي الخاطئ False Negatives غير محتمل. إذا قال فلتر Bloom، "لم تقم بالزحف خلال هذا الموقع"، فهذا يعني أنك بالتأكيد لم تزحف خلال هذا الموقع.

مُرشِّحات - فلاتر Bloom رائعة لأنها تشغل مساحة Space صغيرة جدًا. فجدول التجزئة Hash Table يجب أن يخزن كل عنوان URL تم الزحف خلاله Crawled بواسطة Google، ولكن لا يتعين على فلتر Bloom Filter القيام بذلك. إنها رائعة عندما لا تحتاج إلى إجابة دقيقة Exact Answer، كما هو الحال في كل هذه الأمثلة. لا يوجد مشكلة في أن يقول موقع bit.ly، "نعتقد أن هذا الموقع Site قد يكون ضارًا Malicious، لذا توخ مزيدًا من الحذر"

خوارزميات HyperLogLog

على نفس المنوال توجد خوارزمية Algorithm أخرى تسمى HyperLogLog. لنفترض أن Google تريد حساب عدد عمليات البحث الفريدة Unique Searches التي يقوم بها مستخدموها Users. أو افترض أن أمازون Amazon تريد حساب عدد العناصر الفريدة Unique Items التي نظر إليها المستخدمون Users اليوم. الإجابة على هذه الأسئلة تأخذ مساحة Space كبيرة! مع Google، يجب عليك الاحتفاظ بسجل Log لجميع عمليات البحث الفريدة Unique Searches. عندما يبحث مستخدم عن شيء ما، عليك معرفة ما إذا كان موجودًا بالفعل في السجل Log. إذا لم يكن كذلك، يجب عليك إضافته إلى السجل. حتى في يوم واحد، سيكون هذا السجل Log ضخمًا جداً!

تقوم خوارزمية HyperLogLog بتقريب Approximate عدد العناصر الفريدة Unique Elements في مجموعة Set. تمامًا مثل فلاتر – مرشحات بلوم Bloom Filters، لن تمنحك إجابة دقيقة، لكنها تقترب جدًا منها وتستخدم جزءًا صغيرًا فقط من الذاكرة Memory في مهمة Task مثل هذه.

إذا كان لديك الكثير من البيانات Data وتُرضيك الإجابات التقريبية Approximate Answers، فتحقق من الخوارزميات الاحتمالية Probabilistic Algorithms!

خوارزميات SHA

هل تتذكر التجزئة Hashing من الفصل الخامس؟ فقط للتلخيص، لنفترض أن لديك مفتاحًا Key، وتريد وضع القيمة المرتبطة Associated Value به في مصفوفة Array.

يمكنك استخدام دالة تجزئة Hash Function لإخبارك بالخانة Slot التي يجب وضع القيمة Value فيها.

وتضع القيمة Value في تلك الخانة Slot.

يتيح لك ذلك إجراء عمليات بحث بوقت ثابت Constant-Time Lookups. عندما تريد معرفة قيمة Value مفتاح Key، يمكنك استخدام دالة التجزئة Hash Function مرة أخرى، وسوف تخبرك في وقت $O(1)$ بالخانة Slot التي يجب التحقق منها.

في هذه الحالة، تريد أن تمنحك دالة التجزئة توزيعًا Distribution جيدًا. لذلك تأخذ دالة التجزئة Hash Function سلسلة حروف String وتعيد لك رقم الخانة Slot Number لتلك السلسلة String.

مقارنة الملفات Comparing Files

دالة تجزئة Hash Function أخرى هي دالة خوارزمية التجزئة الآمنة (SHA Function) Secure Hash Algorithm Function. بإعطائها سلسلة حروف String، تُعطيك دالة SHA تجزئة Hash لتلك السلسلة String.

يمكن أن تكون المصطلحات Terminology محيرة بعض الشيء هنا. SHA هي دالة تجزئة Hash Function. تُولِّد تجزئة Generate Hash، وهي مجرد سلسلة حروف قصيرة Short String. تنتقل دالة التجزئة Hash

Function لجداول التجزئة Hash Tables من سلسلة حروف String إلى فهرس مصفوفة Array Index، بينما تنتقل دالة SHA من سلسلة حروف String إلى سلسلة حروف String.

تنشئ دالة SHA تجزئة Hash مختلفة لكل سلسلة حروف String.

ملاحظة

تجزئات SHA Hashes طويلة. لقد تم اقتطاعها هنا.

يمكنك استخدام SHA لمعرفة ما إذا كان ملفان Two Files متطابقان. يكون هذا مفيدًا عندما يكون لديك ملفات كبيرة جدًا. افترض أن لديك ملف حجمه 4 جيجا بايت. تريد التحقق مما إذا كان لدى صديقك نفس هذا الملف الكبير. لست مضطرًا لمحاولة إرسال ملفك الكبير إليه عبر البريد الإلكتروني Email. بدلاً من ذلك، يمكنك حساب Calculate تجزئة SHA Hash ومقارنتها.

التحقق من كلمات المرور Checking Passwords

تُعد SHA مفيدة أيضًا عندما تريد مقارنة سلاسل الحروف Compare Strings دون الكشف عن ماهية سلسلة الحروف الأصلية Original String. على سبيل المثال، افترض أن Gmail تعرض للاختراق Hacked، وأن المهاجم سرق كل كلمات المرور Passwords! هل تصبح كلمة المرور الخاصة بك في العراء؟ لا، ليس كذلك. لا تُخزِّن Google كلمة المرور الأصلية Original Password، تقوم فقط بتخزين تجزئة SHA Hash لكلمة المرور! عندما تكتب كلمة المرور الخاصة بك، يقوم Google بتجزئتها Hash والتحقق Checks منها مقابل Against التجزئة Hash في قاعدة البيانات Database الخاصة بها.

لذلك فهي تقارن فقط علامات التجزئة Hashes - ليس من الضروري تخزين Store كلمة مرورك Password! يتم استخدام SHA بشكل شائع جدًا لتجزئة كلمات المرور Hash Passwords مثل هذا. إنها تجزئة ذو اتجاه واحد One-Way Hash. يمكنك الحصول على تجزئة Hash لسلسلة حروف String.

لكن لا يمكنك الحصول على سلسلة الحروف الأصلية Original String من التجزئة Hash.

وهذا يعني أنه إذا حصل المهاجم على تجزئات SHA Hashes من Gmail، فلن يتمكن من تحويل Convert علامات التجزئة Hashes هذه مرة أخرى إلى كلمات المرور الأصلية Original Passwords! يمكنك تحويل كلمة المرور إلى تجزئة، ولكن ليس العكس.

SHA هي في الواقع عائلة من الخوارزميات Algorithms Family: SHA-0 و SHA-1 و SHA-2 و SHA-3.
حتى كتابة هذه السطور، يوجد بعض نقاط الضعف Weaknesses في SHA-0 وSHA-1. إذا كنت تستخدم خوارزمية SHA لتجزئة كلمة المرور Password Hashing، فاستخدم SHA-2 أو SHA-3. المعيار الذهبي Gold Standard لدوال تجزئة كلمة المرور Password-Hashing Functions حاليًا هو bcrypt (على الرغم من أنه لا يوجد شيء مضمون بشكل كامل Foolproof).

التجزئة الحساسة للموضع Locality-Sensitive Hashing

تتمتع SHA بملمح Feature آخر مهم: أنها غير حساسة للموضع Locality-Insensitive. لنفترض أن لديك سلسلة حروف String، وتقوم بإنشاء Generate تجزئة Hash لها.

إذا غيرت حرفًا Character واحدًا فقط من سلسلة الحروف String وأعدت إنشاء Regenerate التجزئة Hash، فستكون مختلفة تمامًا!

يعد هذا أمرًا جيدًا لأن المهاجم لا يمكنه مقارنة علامات التجزئة Compare Hashes لمعرفة ما إذا كان قريبًا من اختراق كلمة المرور Cracking Password.
في بعض الأحيان، تريد العكس: تريد دالة تجزئة حساسة للموضع Locality-Sensitive Hash Function. وهنا يأتي دور دالة Simhash. إذا أجريت تغييرًا بسيطًا على سلسلة حروف String، فإن دالة Simhash تُولِّد Generate تجزئة Hash مختلفة قليلاً فقط. يتيح لك هذا مقارنة علامات التجزئة Compare Hashes ومعرفة مدى تشابه سلسلتين Strings، وهو أمر مفيد جدًا!

- يستخدم Google دالة Simhash لاكتشاف التكرارات Duplicates أثناء الزحف Crawling خلال الويب Web.
- يمكن للمدرس استخدام Simhash لمعرفة ما إذا كان الطالب يقوم بنسخ مقال من الويب.
- يسمح موقع Scribd للمستخدمين Users برفع Upload المستندات Documents أو الكتب لمشاركتها مع الآخرين. لكن Scribd لا يريد للمستخدمين رفع محتوى محمي بحقوق الطبع والنشر Copyrighted Content! يمكن للموقع استخدام Simhash للتحقق مما إذا كان الرفع مشابهًا لكتاب Harry Potter، وإذا كان الأمر كذلك، فيقوم برفضه تلقائيًا Reject Automatically.

Simhash مفيدة عندما تريد التحقق Check من وجود عناصر مشابهة Similar Items.

تبادل مفاتيح ديفي-هيلمان Diffie-Hellman Key Exchange

تستحق خوارزمية Diffie-Hellman أن تُذكر هنا، لأنها تحل مسألة قديمة بطريقة أنيقة. كيف تقوم بتشفير Encrypt رسالة Message بحيث يمكن قراءتها فقط من قبل الشخص الذي أرسلت إليه الرسالة؟

أسهل طريقة هي التوصل إلى شفرة Cipher، مثل a = 1، b = 2، وهكذا. ثم إذا أرسلت لك الرسالة "4,15,7"، يمكنك ترجمتها إلى "d,o,g". ولكن لكي ينجح هذا، علينا أن نتفق Agree على الشفرة Cipher. لا يمكننا الاتفاق عبر البريد الإلكتروني Email، لأن شخصًا ما قد يخترق Hack بريدك الإلكتروني ويكتشف الشفرة ويفك تشفير Decode رسائلنا. حتى لو التقينا وجهًا لوجه، فقد يخمن شخص ما الشفرة - فالأمر ليس معقدًا. لذلك يجب علينا تغييرها كل يوم. ولكن عند ذلك يجب أن نلتقي شخصيًا لتغييرها كل يوم!

حتى لو تمكنا من تغييرها كل يوم، فمن السهل كسر Crack شفرة بسيطة كهذه بهجوم القوة الغاشمة Brute-Force Attack. افترض أنني رأيت الرسالة "9,6,13,13,16 24,16,19,13,5". سوف أخمن أن هذا يستخدم a = 1، b = 2، وهكذا.

هذا هراء. لنجرب a = 2، b = 3، وهكذا.

هل تظن أن هذا ينجح! من السهل كسر شفرة بسيطة مثل هذه. استخدم الألمان شفرة أكثر تعقيدًا في الحرب العالمية الثانية، لكن تم كسرها Cracked برغم ذلك. تقوم خوارزمية Diffie-Hellman بحل كلا المسألتين:

- كلا الطرفين لا يحتاج إلى معرفة الشفرة Cipher. لذلك ليس علينا أن نلتقي ونتفق على ما يجب أن تكون عليه الشفرة.
- من الصعب للغاية فك تشفير Decode الرسائل المشفرة Encrypted Messages.

لدى Diffie-Hellman مفتاحان Two Keys: مفتاح عام Public Key ومفتاح خاص Private Key. المفتاح العام يعني بالضبط: عام Public. يمكنك نشره على موقع الويب الخاص Website بك أو إرساله بالبريد الإلكتروني إلى الأصدقاء أو القيام بأي شيء تريده به. لا داعي لإخفائه. عندما يريد شخص ما إرسال رسالة إليك، يقوم بتشفيرها Encrypt باستخدام المفتاح العام Public Key. لا يمكن فك تشفير Decrypt الرسالة المشفرة Encrypted Message إلا باستخدام المفتاح الخاص Private Key. طالما أنك الشخص الوحيد الذي لديه المفتاح الخاص، فلن يتمكن أحد سواك من فك تشفير هذه الرسالة!

لا تزال خوارزمية Diffie-Hellman مستخدمة في الممارسة، جنبًا إلى جنب مع خليفتها، خوارزمية RSA. إذا كنت مهتمًا بالتشفير Cryptography، فإن خوارزمية Diffie Hellman هي مكان جيد للبدء: إنها أنيقة Elegant ولا يصعب متابعتها.

البرمجة الخطية Linear Programming

لقد أبقيت الأفضل للنهاية. البرمجة الخطية هي واحدة من أروع الأشياء التي أعرفها.

تُستخدم البرمجة الخطية Linear Programming لتعظيم Maximize شيء في ظل بعض القيود Constraints. على سبيل المثال، افترض أن شركتك تصنع منتجين، القمصان وحقائب اليد. القمصان تحتاج 1 متر من القماش و5 أزرار. وتحتاج حقائب اليد إلى مترين من القماش وزرين. لديك 11 مترًا من القماش و20 زرًا. أنت تكسب 2 دولار لكل قميص و3 دولارات لكل حقيبة يد. كم عدد القمصان وحقائب اليد التي يجب عليك صنعها لزيادة أرباحك إلى أقصى حد؟

هنا تحاول تعظيم Maximize الربح، وأنت مُقيَّد Constrained بكمية الخامات التي لديك.

مثال آخر: أنت سياسي وتريد تعظيم Maximize عدد الأصوات التي تحصل عليها. أظهر بحثك أن الأمر يستغرق في المتوسط ساعة من العمل (تسويق Marketing، بحث Research، وما إلى ذلك) لكل صوت من سكان سان فرانسيسكو أو 1.5 ساعة لكل صوت من سكان شيكاغو. أنت بحاجة إلى ما لا يقل عن 500 من سكان سان فرانسيسكو و300 من سكان شيكاغو. لديك 50 يومًا. كما أنه يكلفك 2 دولار لكل صوت من سكان سان فرانسيسكو مقابل 1 دولار لكل صوت من سكان شيكاغو. إجمالي ميزانيتك 1500 دولار. ما هو الحد الأقصى Maximum لعدد الأصوات الإجمالية التي يمكنك الحصول عليها من (سان فرانسيسكو + شيكاغو)؟ هنا تحاول تعظيم Maximize الأصوات، وأنت مقيد Constrained بالوقت والمال.

قد تفكر، "لقد تحدثت عن الكثير من موضوعات التحسين Optimization في هذا الكتاب. كيف هي مرتبطة بالبرمجة الخطية Linear Programming؟ " يمكن إجراء جميع خوارزميات الرسم البياني Graph Algorithms من خلال البرمجة الخطية Linear Programming بدلاً من ذلك. البرمجة الخطية هي إطار عمل

عام General Framework أكثر من أي شيء آخر، وتعد مسائل الرسم البياني Graph Problems مجموعة فرعية Subset من ذلك. أتمنى أن يتفتَّح عقلك!

تَستخدم البرمجة الخطية Linear Programming خوارزمية سيمبلكس Simplex Algorithm. إنها خوارزمية معقدة Complex Algorithm، ولهذا لم أقم بتضمينها في هذا الكتاب. إذا كنت مهتمًا بالتحسين Optimization، فابحث عن البرمجة الخطية Linear Programming!

الخاتمة

آمل أن توضح لك هذه الجولة السريعة المكونة من 10 خوارزميات Algorithms مقدار المزيد المتبقي لاكتشافه. أعتقد أن أفضل طريقة للتعلم هي العثور على شيء تهتم به والتعمق فيه. أعطاك هذا الكتاب أساسًا متينًا للقيام بذلك.

Coding عربي

الإجابات على التمارين
Answers To Exercises

الفصل الأول

1.1 الإجابة: 7

1.2 الإجابة: 8

1.3 الإجابة: O(log n)

1.4 الإجابة: O(n)

1.5 الإجابة: O(n)

1.6 الإجابة: O(n). قد تفكر، "أنا أفعل هذا فقط لحرف Character واحد من 26 حرفًا، لذا يجب أن يكون وقت التشغيل Run Time O(n÷26)." إحدى القواعد البسيطة التي يجب تذكرها، تجاهل الأرقام Numbers التي يتم إضافتها Added أو طرحها Subtracted أو ضربها Multiplied أو قسمتها Divided. ليس أيًا من هذه أوقات تشغيل Big O صحيحة:
O(n + 26)، O(n - 26)، O(n * 26)، O(n / 26). كلهم مثل O(n)! لماذا؟ إذا كنت فضوليًا، فانتقل إلى "إعادة النظر في تدوين Big O" في الفصل الرابع، واقرأ الثوابت Constants في تدوين Big O (الثابت Constant هو مجرد رقم Number؛ كان 26 هو الثابت Constant في هذا السؤال).

الفصل الثاني

2.1 الإجابة: في هذه الحالة، تقوم بإضافة Adding المصروفات إلى القائمة List كل يوم وقراءة Reading جميع النفقات مرة واحدة في الشهر. تتمتع المصفوفات Arrays بقراءات سريعة Fast Reads وإدخالات - إدراجات بطيئة Slow Inserts. القوائم المرتبطة Linked Lists لديها قراءات بطيئة Slow Reads وإدخالات سريعة Fast Inserts. نظرًا لأنك ستقوم بالإدراج Inserting غالبا أكثر من

القراءة Reading، فمن المنطقي استخدام قائمة مرتبطة Linked List. بالإضافة إلى ذلك، فإن القوائم المرتبطة Linked Lists تكون قراءاتها بطيئة Slow Reads فقط إذا كنت تحاول الوصول Accessing إلى عناصر عشوائية Random Elements في القائمة List. نظرًا لأنك تقرأ كل عنصر في القائمة، فإن القوائم المرتبطة Linked Lists ستحقق نتائج جيدة في عمليات القراءة Reads أيضًا. لذا فإن القائمة المرتبطة هي حل Solution جيد لهذه المسألة Problem.

2.2 الإجابة: قائمة مرتبطة Linked List. تحدث الكثير من الإدخالات Inserts (تقوم الخوادم Servers بإضافة الطلبات Add Orders)، والتي تتفوق فيها القوائم المرتبطة Linked Lists. لا تحتاج إلى البحث Search أو الوصول العشوائي Random Access (هذا ما تتفوق فيه المصفوفات Arrays)، لأن الطهاة يقومون دائمًا بنزع الطلب الأول من قائمة الانتظار Queue.

2.3 الإجابة: مصفوفة مرتبة Sorted Array. تمنحك المصفوفات Arrays وصولاً عشوائيًا Random Access - يمكنك الحصول على عنصر من منتصف المصفوفة على الفور Instantly. لا يمكنك فعل ذلك مع القوائم المرتبطة Linked Lists. للوصول إلى العنصر الأوسط في قائمة مرتبطة، يجب أن تبدأ من العنصر الأول وتقوم باتباع Follow جميع الروابط Links وصولاً إلى العنصر الأوسط Middle.

2.4 الإجابة: الإدراج - الإدخال Inserting في المصفوفات Arrays بطيء. أيضًا، إذا كنت تستخدم البحث الثنائي Binary Search للبحث عن أسماء المستخدمين Usernames، فيجب ترتيب Sort المصفوفة. افترض أن شخصًا ما يُدعى Adit B اشترك Sign Up في Facebook. سيتم إدراج Insert اسمه في نهاية المصفوفة. لذلك تحتاج إلى ترتيب المصفوفة في كل مرة يتم فيها إدخال Insert اسم!

2.5 الإجابة: البحث Searching — أبطأ من المصفوفات Arrays، أسرع من القوائم المرتبطة Linked Lists. الإدراج Inserting — أسرع من المصفوفات Arrays، نفس مقدار الوقت مثل القوائم المرتبطة Linked Lists. لذلك فإنه أبطأ في البحث Searching عن المصفوفة Array، ولكنه أسرع أو مماثل لسرعة القوائم المرتبطة Linked Lists لكل شيء. سنتحدث عن هيكل بيانات مختلط Hybrid Data Structure آخر يسمى جدول التجزئة Hash Table لاحقًا في الكتاب. يجب أن يمنحك هذا فكرة عن كيفية بناء هياكل بيانات Data Structures أكثر تعقيدًا من الهياكل البسيطة.

إذن ما الذي يستخدمه Facebook حقًا؟ من المحتمل أنه يستخدم عشرات قواعد البيانات Databases المختلفة، مع هياكل بيانات Data Structures مختلفة وراءها: جداول التجزئة Hash Tables، أشجار B-Trees، وغيرها. المصفوفات Arrays والقوائم المرتبطة Linked Lists هي اللبنات الأساسية Building Blocks لهياكل البيانات Data Structures هذه الأكثر تعقيدًا.

الفصل الثالث

3.1 الإجابة: فيما يلي بعض الأشياء التي يمكن أن تخبرني بها:

- يتم استدعاء Call دالة `greet` أولاً، مع `name = maggie`.
- ثم تقوم دالة `greet` باستدعاء Call دالة `greet2`، مع `name = maggie`.
- في هذه المرحلة، تكون دالة `greet` في حالة معلَّقة غير مكتملة Incomplete Suspended State.
- استدعاء الدالة الحالي Current Function Call هو دالة `greet2`.
- بعد اكتمال استدعاء الدالة Function Call هذا، سيتم استئناف دالة `greet`.

3.2 الإجابة: الدفتر Stack ينمو إلى الأبد Grows Forever. كل برنامج Program لديه مقدار محدود من المساحة Space على دفتر الاستدعاءات Call Stack. عندما تنفد مساحة برنامجك (وهو ما سينتهي به الأمر في النهاية)، سيقوم بالخروج Exit مع وجود خطأ في امتلاء الدفتر Stackoverflow.

الفصل الرابع

10.4 الإجابة:

```
def sum(list):
  if list == []:
    return 0
  return list[0] + sum(list[1:])
```

10.5 الإجابة:

```
def count(list):
  if list == []:
    return 0
  return 1 + count(list[1:])
```

10.6 الإجابة:

```
def max(list):
  if len(list) == 2:
    return list[0] if list[0] > list[1] else list[1]
  sub_max = max(list[1:])
  return list[0] if list[0] > sub_max else sub_max
```

10.7 الإجابة: الحالة الأساسية Base Case للبحث الثنائي Binary Search هي مصفوفة Array تحتوي على عنصر Item واحد. إذا كان العنصر الذي تبحث عنه يطابق Match العنصر الموجود في المصفوفة، فقد عثرت عليه! خلاف ذلك، فإنه ليس في المصفوفة.

في الحالة التكرارية Recursive Case للبحث الثنائي Binary Search، تقوم بتقسيم Split المصفوفة إلى نصفين، والتخلص من النصف، واستدعاء Call البحث الثنائي Binary Search علي النصف الآخر.

Coding عربي

4.5 الإجابة: $O(n)$

4.6 الإجابة: $O(n)$

4.7 الإجابة: $O(1)$

4.8 الإجابة: $O(n^2)$

الفصل الخامس

5.1 الإجابة: متسقة Consistent

5.2 الإجابة: غير متسقة Not Consistent

5.3 الإجابة: غير متسقة Not Consistent

5.4 الإجابة: متسقة Consistent

5.5 الإجابة: دوال التجزئة (Hash Functions) C و D ستعطي توزيعًا Distribution جيدًا.

5.6 الإجابة: دوال التجزئة B و D ستعطي توزيعًا Distribution جيدًا.

5.7 الإجابة: دوال التجزئة B و C و D ستعطي توزيعًا Distribution جيدًا.

الفصل السادس

6.1 الإجابة: أقصر مسار Shortest Path يبلغ طوله 2.

6.2 الإجابة: أقصر مسار Shortest Path يبلغ طوله 2.

6.3 الإجابات: A - غير صالح Invalid؛ B- صالح Valid؛ C - غير صالح Invalid.

6.4 الإجابة: 1 – استيقظ Wake Up 2 - التمرين Exercise 3 – الاستحمام Shower 4 – تنظيف الأسنان Brush Teeth 5 - ارتداء الملابس Get Dressed 6 - تعبئة الغداء Pack Lunch 7 - تناول الفطور Eat Breakfast.

6.5 الإجابات: A - شجرة Tree؛ B ـ ليست شجرة Not a Tree؛ C ـ شجرة Tree. المثال الأخير هو مجرد شجرة جانبية. الأشجار Trees هي مجموعة فرعية Subset من الرسوم البيانية Graphs. لذا فإن الشجرة Tree هي دائمًا رسم بياني Graph، لكن الرسم البياني Graph قد يكون أو لا يكون شجرة Tree.

الفصل السابع

7.1 الإجابات: A: A - 8؛ B - 60؛ C ـ سؤال مخادع. لا يوجد أقصر مسار Shortest Path محتمل (دورة الوزن - الترجيح السالب Negative-Weight Cycle).

الفصل الثامن

8.1 الإجابة: تتمثل الإستراتيجية الشرهة - الطامعة Greedy Strategy في اختيار أكبر صندوق Box يلائم Fit المساحة المتبقية، وتكرار ذلك حتى لا تتمكن من تعبئة المزيد من الصناديق. لا، هذا لن يمنحك الحل الأمثل Optimal Solution.

8.2 الإجابة: استمر في اختيار النشاط ذو أعلى قيمة نقطية Point Value والتي لا يزال بإمكانك القيام به في الوقت المتبقي. توقف عندما لا تستطيع فعل أي شيء آخر. لا، هذا لن يمنحك الحل الأمثل.

8.3 الإجابة: لا

8.4 الإجابة: نعم

8.5 الإجابة: نعم

8.6 الإجابة: نعم

8.7 الإجابة: نعم

8.8 الإجابة: نعم

الفصل التاسع

9.1 الإجابة: نعم. ثم يمكنك سرقة MP3 Player و IPhone و Guitar بقيمة إجمالية قدرها 4500 دولار.

9.2 الإجابة: يجب أن تأخذ الماء Water والطعام Food والكاميرا Camera.

9.3 الإجابة:

	C	L	U	E	S
B	0	0	0	0	0
L	0	1	0	0	0
U	0	0	2	0	0
E	0	0	0	3	0

10.1 الإجابة: يمكنك استخدام شيء يسمى التسوية Normalization. تنظر إلى متوسط التقييم Average Rating لكل شخص وتستخدمه لتوسيع نطاق Scale تقييماتهم. على سبيل المثال، قد تلاحظ أن متوسط تقييم Pinky هو 3، بينما متوسط تقييم Yogi هو 3.5. لذا، فإنك ترفع تقييمات Pinky قليلاً، حتى يصل متوسط تقييمها إلى 3.5 أيضًا. ثم يمكنك مقارنة تقييماتهم على نفس المقياس Scale.

10.2 الإجابة: يمكنك إعطاء وزن - ترجيح Weight أكبر لتقييمات Ratings المؤثرين Influencers عند استخدام KNN. افترض أن لديك ثلاثة جيران Neighbors: Joe و Dave و Wes Anderson (شخصية مؤثرة Influencer). قاموا بتقييم Caddyshack على 3 a و 4 a و 5 a على التوالي. بدلاً من مجرد أخذ متوسط تقييماتهم (3 + 4 + 5 / 3 = 4 Stars)، يمكنك منح تقييم Wes Anderson وزنًا Weight أكبر: 3 + 4 + 5 + 5 + 5 / 5 = 4.4 Stars.

10.3 الإجابة: إنه منخفض للغاية Too Low. إذا نظرت إلى عدد أقل من الجيران Neighbors، فهناك احتمال أكبر بأن تكون النتائج مشوهة. من القواعد الأساسية الجيدة أنه إذا كان لديك عدد N من المستخدمين Users، فيجب أن تنظر إلى جيران عددهم يساوي sqrt(N) وهو الجذر التربيعي لـ N.

Coding عربي