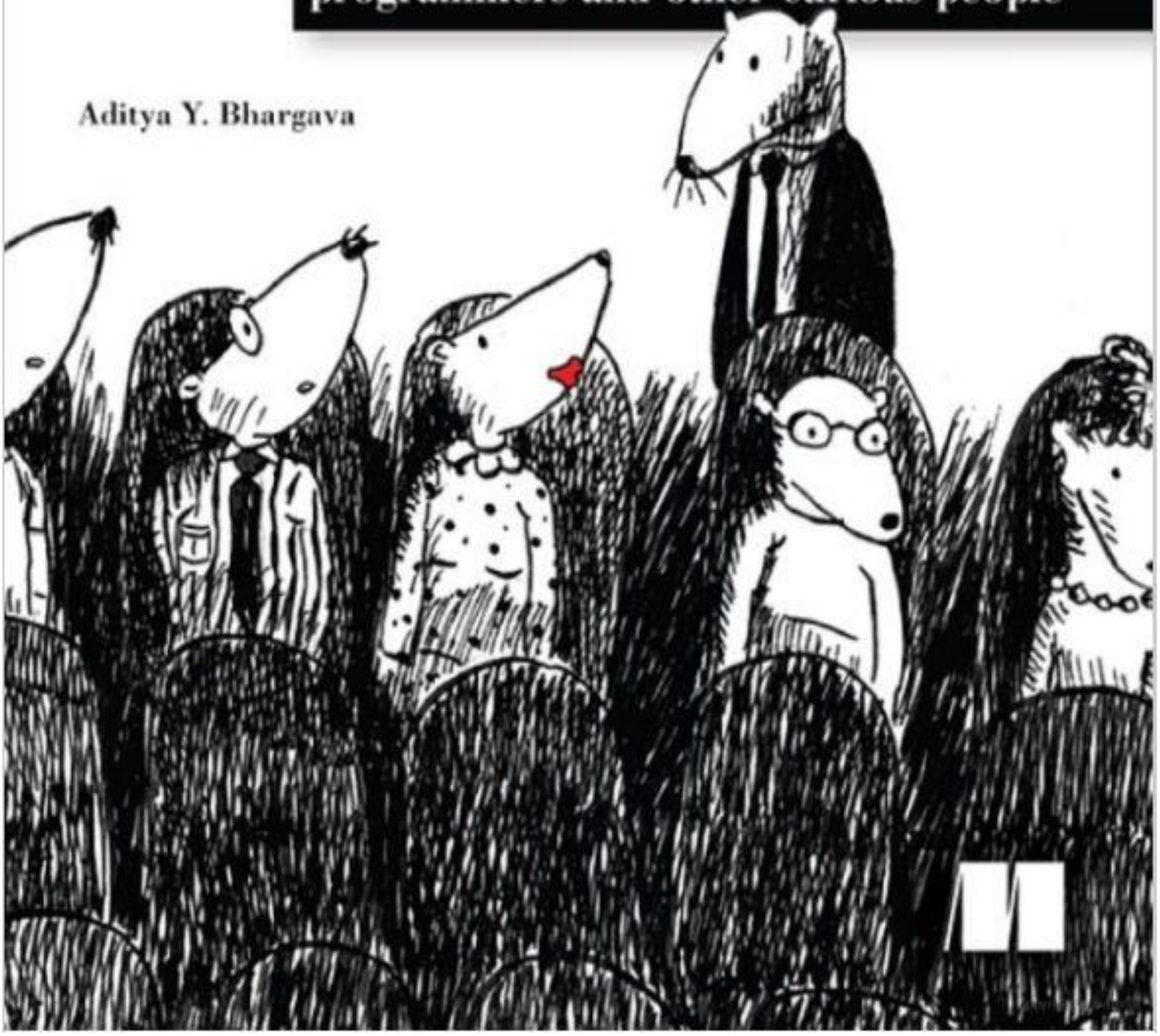
grokking algorithms

An *illustrated* guide for programmers and other curious people

Aditya Y. Bhargava



Intro

إن شاء الله ده هيكون ملخص سريع لكل شابتر من كتاب " Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People " بما إنه كتاب شيق جدا وسهل القرائة وسريع، احنا كنا ملخصين كل شابتر في شكل مقالات على LinkedIn الخاص بي Moustafa Shomer.
و من اللي عرفته إن الكتاب مخصص للمبتدئين لسه في ال Algorithms -أنا- .
فأتمنى إن محدش ينزعج من الطريقة البسيطة و يحاول ياخدها ك refresher حتى لو هو دارس المادة بتاعتها في الجامعة أو قارئ كتب مختصة في الموضوع.
و كل الشكر لزميلتي Menna Nawar لمراجعتها و مساندتها الدائمة في الكتابة و النشر، الكتاب ده طلع للنور بسبب جميع القراء اللي أبدوا شغف واهتمام للي بنعمله، ف نتمنى تستفيدوا و يعجبكم و تنشروه ك صدقة علم.

Chapter 1

في أول شابتر الكاتب بيقارن بين Binary search و Simple search
و هو باختصار انك لو بتحاول تدور على اسم حد على فيسبوك مثلا ف عندك طريقتين:
1- إنك تعدي على اسم اسم فالداتا بتاعة فيسبوك الي عليها كل الحسابات لحد متوصل للإسم الي انت عاوزه
2- إنك ترتب الأسامي دي أبجديا و بعدها تدور بطريقة متكررة عن أقرب حروب بتمثل الإسم الي بتدور عليه
طبعا أول طريقة في حالة ملايين الداتا هتاخد ملايين العمليات, لانك في كل عملية بتقارن الإسم الي إنت عاوزه بالإسم الي طلعلك, في تاني عملية بيعتمد على ترتيب الداتا لسبب مهم جدا.
و خطوات عمل الألجورذم كالاتي:
1- بتحدد قيم max - min و دي بتعتمد على الداتا بتاعتك
2- بتختار القيمة الي فالنص و بتقارنها بالقيمة الي بتدور عليها
3- لو القيمة الي فالنص أكبر من القيمة الي انت عاوزها يبقى هتخليها ال max الجديدة بتاعتك
و لو القيمة الي فالنص دي أقل من القيمة الي بتدور عليها يبقى هتخليها هي ال min الجديدة
4- هترجع تاني لنقطة 2
و هتفضل تلف في ال loop دي لحد متلاقي القيمة الي بتدور عليها دي, سواء رقم معين وسط أرقام كتير - مترتبة- ، أو مش هتلاقي القيمة دي و ساعتها المفروض الألجورذم بيطلعلك null إن القيمة دي مش موجودة أصلا
الفايدة فالموضوع انك لو عندك data point 240000 مثلا و هتقارن عليها السيرش بتاعك, و قول انك في أسوأ حالة هتضطر في حالة الألجورذم البسيط انك تعمل 240000 خطوة لحد متوصل للرقم بتاعك, أما في حالة ال Binary search ف هتحتاج 18 خطوة فقط عشان توصل للحل.
هو برضو اتكلم عن كيفية حساب سرعة الألجورذمات باستخدام الBig O notation، ولكن هنتكلم عنها بالتفصيل في الشباتر الجاية.

Chapter 2

في الشابتر ده الكاتب بيتكلم عن ال linked lists vs arrays وبيقول ال pros/cons لكل واحد
في الأول هو اتكلم عن الميموري وازاي البيانات بتتخزن فيها يا اما بطريقة منظمة ورا بعض، يا اما بطريقة مش منظمة ورا بعض لكن كل مكان بيشير للي بعده

الlinked list:

هنا الداتا بتكون مترتبة في أي مكان في الميموري، وكل مكان بيشير للمكان اللي بعده اللي فيه داتا
من مزاياها انك بتقدر تعمل insertion بشكل سريع
من عيوبها انك بتاخد وقت أطول عشان تقرا داتا، لأنك محتاج تعدي على كل الأماكن البي قبليها عشان تعرف توصلها، يعني لو الداتا ترتيبها العاشرة، هتحتاج تمشي من رقم واحد عشان توصل لاتنين ومن اتنين لتلاتة وهكذا.

الarrays:

هنا الداتا بتكون منظمة بالترتيب ورا بعضها، لو عندك خمس اماكن في الميموري في الخمسة بيكونوا ورا بعض، و ده بيتيح ليك انك تقرا اي داتا انت عاوزها من غير ما تحفظ مكان الداتا اللي بعدها.
من مزاياها إنك تقدر تقرأ الداتا بسرعة جدا، و بتحتاج ده عشان تنفذ ألجورذمات زي ال binary search الي اتكلمنا عنه فالبوست الي فات
من عيوبها إنك كل مهتحب تزود داتا أو تمسح داتا محتاج تغير ال تركيب بتاع الميموري الي محفوظ فيه الداتا.
بمعنى انت لو عندك ٥ خانات للداتا بيكونوا محفوظين ورا بعض، لو احتاجت تضيف خانة زيادة ف انت هنا محتاج خانة جمبهم، و لو الخانة دي في الميموري مشغولة، يبقى هتضطر تدور على ٦ خانات تانيين ورا بعض في الميموري، وهكذا كل ما تحب تضيف عنصر جديد.
ومن الحلول المطروحة للمشكلة دي انك تقدر بكل بساطة تحجز مثلا ١٠ خانات مسبقا بحيث اي عنصر جديد هيدخل في الخانات الفاضية، بس عيب ده انك بتحجز مكان فالميموري ممكن متستغلهوش.
باستخدام الاتنين دول نقدر نعمل أغلب الداتا ستراكتشرز اللي بنستخدمها في أغلب ال applications ولغات البرمجة الحالية، زي اننا نقدر نعمل array of linked lists وهكذا.
في الاخر اتكلم عن ال selection sort وهو انك بتحاول ترتب داتا عندك وبتضطر تلف في كل عنصر فيها وتقارنه بباقي العناصر اللي قبلها و تاخد أقل-أعلى قيمة و تحطها في list-array جديدة، و في الشابتر الجاي هيتكلم عن ألجورذم أسهل وأسرع منه اللي هوquick sort.

Chapter 3

في أول الشابتر الكاتب شرح ال recursive functions وهي أي فنكشن بتنادي على نفسها تاني أو بتعمل call لنفسها يعني.
زي ايه مثلا؟
مثلا لو عاوز اكتب فنكشن تفضل تزود الرقم بقيمة ١ لحد متوصل للرقم المطلوب، فلو بدأنا بصفر هتفضل تزود واحد لحد متوصل ل y وهو اللي انت عاوزة.
طب ايه المشكلة ما نعملها في فور لووب عادية؟
الفكرة ان عشان تعمل clean code مفهوم و سهل يتقرأ الأحسن من انك تكتب for loop بتعمل جواه عمليات كتير، انك تعمل فنكشن وتحط جواها العمليات دي كلها، بس بطبيعة الحال أحيانا بتضطر تعملها for loop لو بقى فيه مشاكل فالmemory.
طبعا المثال كمان بسيط جدا للتوضيح مش اكتر.
الفكرة بقى هنا ان عشان تعمل recursion صح لازم شرط مهم جدا، و هو انك متنساش ابدا تحط الشرط الأساسي الي عاوز توصله.
مثلا لو عاوز فنكشن تزود واحد على الرقم اللي جي كل مرة، ابسط حاجة x=x+1 وخلاص، هنا اللي داخل للفنكشن x، طب والفنكشن المفروض هتعرف منين انها وصلت للرقم المطلوب؟ لازم تحط if condition بقى، if x = y و ده الشرط اللي على أساسه بتطلع من الrecursion loop، و طبعا تخيل لو معملتهوش فالفنكشن هتفضل تاخد اكس تزود عليه واحد و تبعته تاني تزود واحد وهكذا.
للتوضيح اكتر الفنكشن شكلها:

```
Def rec(x):
   x= x+1
   rec(x)
```

و دي اللي هتعمل recursion اه بس للأبد، الفكرة انك تكتبها صح كدا:

```
def Rec(X):
   if X == y:
     return X

   else:
    X+=1
    Rec(x)
```

و هنا الكاتب قسمهم إلى:
Base case: وهي الحالة اللي بتعبر عن انك وصلت للمطلوب خلاص و تطلعك برة اللوب.
Recursive case: و دي تقدر تعتبرها الفنكشن بقى اللي بتكررها، انك تزود واحد فالحالة دي.

في الجزء التاني من الشابتر اتكلم عن الstack، و هي يعتبر اي حد اخد مقدمة عن الكمبيوتر ساينس عارف ايه هو الستاك، هو زي صندوق بتحط فيه كتب، معندكش غير فتحة واحدة بس بتحك فيها و بتطلع منها، الكتاب اللي على الوش هو اللي هيطلع الأول، و الكتاب اللي هتدخله جديد هيبقى هو أول واحد على الوش.
الفكرة هنا ربطه للستاك فالبايثون، ودي حاجة مكونتش متخيلها الحقيقة، وهو ان في لغة البايثون مفيش حاجة اسمها اعمل ڤاريابل نوعه ستاك، الفكرة ان الكمبيوتر ذات نفسه بيستخدم الستاك وانت شغال في كذا فنكشن، مش لازم حالة ال recursion، لا أي فنكشن جواها فنكشن تانية.
مثلا لو عندك فنكشن بتاخد رقمين و بتضربهم و بعدها تبعت الناتج لفنكشن تانية تقسم الناتج على ٢ وترجعه عشان يتضرب ف مية.

```
Def fn2(z):
  Z=z/2
  Return Z

Def fn1(x,y):
  Z= x*y
  Return fn2(Z)*100
```

ساعتها الميموري اللي بيتخزن فيها قيم المتغيرات في كل فنكشن بتتحفظ في ستاك، في اول حاجة بتدخل بتكون الفنكشن رقم 1 وبيكون محفوظ فيها قيمة الرقمين x , y وبعدها يتضربوا ويتحفظوا في متغير z ، بعدها بيدخل الستاك فنشكن 2 و بيكون فيها قيمة الناتج z،
و زي ما تعلمنا في الستاك أول حاجة هي بتخلص الاول، في الفنكشن 2 اللي بتقسم هتخلص الأول دلوقتي لأنها احدث حاجة، وبعدها الناتج هيرجع للفنكشن رقم 1 عشان يتضرب في ١٠٠.
وهكذا، أنا عارف الموضوع معقد وصعب يتشرح بالكلام بس الكتاب فعلا شارحه بطريقة رائعة برسومات كتير للأسف مش هقدر أنزلها هنا كلها، بس أتمنى يكون وصل ولو ٢٠ في المية من الشرح. 😁

Chapter 4

في الشابتر ده الكاتب بيتكلم عن طريقة الـ divide and conquerأو الD&C
و هي بكل بساطة انك بتقسم المشكلة بتاعتك لمشاكل أصغر فأصغر، لحد متوصل لأبسط مشكلة، و من هنا بتبدأ تحلها و تطلع على الخطوة ال اللي قبلها و تعوض فيها بالي وصلتله و هكذا.
فكرتك بحاجة صح؟ الـ recursion, اللي اتكلمنا عنه في شابتر ٣
هو يعتبر زي تطبيق عليه، انت بيكون عندكbase case ، و بيكون عندكrecursive case ، و غالبا ابسط حالة بتكون لما ميكونش عندك اي عنصر "element" فالarray
و الحالة اللي بتنادي فيها الـ function تاني، هي اللي بتوصلك لل base case دي.
طريقة الـ D&C مستخدمة في حاجات كتير جدًا حوالينا، و أشهر تطبيق ليها هو الquick sort
و هي عملية ترتيب العناصر في أي array بطريقة سريعة و سهلة جدًا، كالآتي:
-أنت بتختار عنصر في الـ list وتعتبره الأساس بتاعك الي بتقارن بيه أو الanchor
-و تقارن كل العناصر الموجودة في ال list بيه، وتقسمهم على حسب أي criteria عاوز ترتب بيها، مثلا لو اكبر يبقوا في list لوحدهم، ولو أصغر يبقوا في الـ list التانية.
-بعدها بترجع للخطوة الأولى تاني، وتختار عنصر من ال list وتقارن باقي العناصر بيه وتفصلهم وهكذا.
-لما يتبقى عنصر واحد في الـ list اللي عندك يبقى هو ده اللي هترجعه.*base case*
و بكده تقدر تقسم المشكلة لمشاكل صغيرة أبسط و تبدأ تحلها بطريقة هرمية، فنفس الوقت بتستخدم التكرار فانك بتنادي نفس الفنكشن بتاعتك على الأجزاء الأبسط من المشكلة.
بعدها الكاتب قارن بين الquick sort - merge sort، اتكلم عن ازاي الـ bigO notation اللي بتدل على عدد الـ operationsاللي بيعملها algorithm معين، مش دايمًا بتعبر عن السرعة، يعني ال merge و quick، الاتنين ليهم O(n log(n)) ، بس مش معنى كده ان الاتنين ليهم نفس السرعة، لأن الـ bigO بيهمل الـ constants اللي ممكن تعبر عن سرعة العمليات دي.
و ادى مثال عن عمليتين، واحدة بتاخد ثانية و التانية عشر ثواني، بس كل واحدة بتعمل عدد n operations، يبقى ساعتها ال bigO بتاعهم هيساويn ، لكن ده مش بيعكس أبدًا سرعة العمليات ولا يعتبر مقياس للسرعة، بل هو مقياس لعدد العمليات فقط.

Chapter 5

في الشابتر ده الكاتب بيتكلم عن ال hash tables و عن استخدامها و أهميتها و مميزاتها، بإختصار ال hash tables هي ال dictionaries، وغالبًا معمولها implementation في أغلب اللغات.
أولًا عشان ندخل في التفاصيل، لازم نتكلم عن ال hash functions، و هي زي / mapping function encoder بيحول أي input ل output معين، في حالة ال hash tables هو index او مكان في list/array.
ال hashing functions دي مهمة جدا و تعتبر حجر الأساس فال hash tables، لان دمج ال hashing functions و list او array بقى عندك داتا ستراكتشر جديد سرعة الوصول للداتا جواه هي هي سرعة اضافة او ازالة عنصر من عناصره، وهي في ال big O notation = O(1) يعني أي عملية بتاخد وحدة من الزمن مش أكتر.
طبعا ده بيعتمد على نوع الhashing function، لان من مميزات و عيوب ال functions دي ان ممكن تختار نوع يعملك collision، وهو ان يبقى في عنصرين عاوزين ياخدوا نفس ال index في ال list بتاعتك، وساعتها ال index ده هيحتوي على عنصرين متشالين ف list تانية، يعني هيبقى عندك ليست أساسية بيشاور على مكان فيها، والمكان ده جواه ليست فيه العنصرين اللي حصل بينهم collision.
مشكلة ال collision بتتحل باستخدام hashing function أحسن، وأغلب ال implementations في لغات البرمجة مستخدمة hashing function كويسة ومش هتضطر تغيرها.
باستخدام ال hash tables بقى زي مقولنا هتقدر تعمل mapping لأي value عندك ب value تانية، مثلا لو عاوز الداتا بتاعة user معين، بدل مهتحفظها في list لا انت ممكن تحفظ اسمه ك key و ال value تبقى هي الداتا، وبكده كل ما تعوز تجيب الداتا هتدخل اسمه لل hashing function و هي هتعملك mapping وتطلعلك ال value المحفوظه في اقصر وقت ممكن.
طبعا نقدر نستخدم خواص ال hashing tables اننا نعمل cache ل information معينة بتتكرر كتير، لو عندك صفحة static مثلا وحاطتها ف button، بدل ما كل ميتضغط على الزرار ده يعمل request للداتا من على سيرفر، لا انت ممكن تحفظها في hash table ولو كان الزرار ده موجود ضمن ال keys يبقى رجع ال value للصفحة دي على طول، وبكده تكون وفرت وقت و processing.

Chapter 6

في الشابتر ده الكاتب اتكلم عن الbreadth-first search وهو searching algorithm بيستخدم في حالة الgraphs، الحقيقة ترتيب الشرح مكانش عاجبني أوي ف هحاول أنظمه شوية هنا بإذن الله.
أول حاجة نتكلم عن ايه هي الgraphs ، هي عبارة عن حلقات متوصلة ببعضها عن طريق خيوط - تخيل معايا كونان و هو بيربط الأحداث ببعض- ، كل حلقة هنسميها node متصلة بالتانية عن طريق edge.
ومن خواص ال graph ان مش لازم كل الnodes تكون متوصلة بكل الnodes التانية، بمعنى ان عادي تلاتة يكونوا صحاب، و اتنين منهم بس يعرفوا شخص رابع، و ساعتها واحد من التلاتة مش هيكون متوصل بالشخص الرابع ده.
الedges ممكن تكون بتعبر عن علاقة من طرف واحد يعني directional و ممكن تكون unidirectional أو زي ما الكاتب اشار ليها undirectional، بمعنى لو أحمد سلف محمد فلوس ف ممكن نعبر عنها باستخدام directional edge، لكن لو محمد و ياسمين متجوزين يبقى هتعبر عن العلاقة دي ب ٢ directional edges و ساعتها تقدر تستغنى عنها و تحطها unidirectional/undirectional و خلاص، و من هنا هيتفهم ان العلاقة واحدة بين الطرفين، ياسمين متجوزة محمد و محمد متجوز ياسمين.

ال tree نوع من أنواع ال graph لكنه بيمشي في اتجاه واحد، يعني لو تخيلت معايا شجرة العائلة هيكون في طرف فيها الاحفاد و فوقيهم الأبناء و فوقيهم الآباء و هكذا، بتكون ماشية بتسلسل زمني، مفيش حاجة اسمها الجد يبقى حفيد -الا لو بتتفرج على dark-.
حلو.. بعد ما عرفنا الgraph، هنبدأ نعرف الألجورذم ده بيعمل ايه، هو بيساعدك تجاوب على سؤالين في أي graph عندك:

١- هل في طريق يربط بين node و التانية؟

٢- ايه هو أقصر طريق يربط بين ال nodes دي؟

ممكن تتخيل مشكلة انك بتحاول توصل لمكان ما، فبتشوف انت لو واقف فالموقف و بتحاول تروح مكان معين، هل في مواصلة تقدر توصلك للمكان اللي انت عاوزه ده؟
ولو فيه مواصلات تقدر توصلك، فايه هو ابسط مواصلة ممكن تاخدها.
فممكن تركب ٣ أتوبيسات عشان توصل للمكان ده، وممكن تركب مشروع وتاكسي، و ممكن تركب مواصلة واحدة بس تنزلك في المكان اللي انت عاوزة، وفي أسوأ الحالات فأنت متواجد في مكان معزول ومش هتقدر تروح للوجهة بتاعتك دي.

تقدر تحول أي مشكلة بتواجهها لgraph، و تطبق عليه الألجورذم ده عشان تجاوب على الأسئلة الي من النوع ده.

ازاي الألجورذم بيشتغل؟
هو شبه الsimple search بسيط جدًا، بتقف عند node معينة ولتكن الموقف اللي انت عنده، و تاخد كل المواصلات الموجودة وتشوف هتوصلك للوجهة بتاعتك ولا لأ، طبعا كل مواصلة هتوديك لnode تانية.
تشوف هل وصلت بعد أول step ؟، لو لأ يبقى هتاخد تاني خطوة و هتركب كل المواصلات المتاحة في المكان اللي انت نزلت فيه ده، و تسأل نفسك تاني وصلت للnode اللي انت عاوزها؟ و هكذا.

وفي Gif بيوضح العملية بالترتيب هنا قبل الشرح: GIF
الألجورذم بيرتب كل الاحتمالات المتاحة و بيمشي في كل فرع مع كل step، اول خطوة دي بتبقى first degree يعني أقرب حاجة، تاني خطوة بتديك ال second degree و هكذا.
لو وصلت لل node اللي انت عاوزها معنى كدا ان اه فيه path between node A and node B، وفي نفس الوقت بما انك ماشي ب steps منتظمة و مرتبة عن طريق الdegree، فانت بيكون عندك كمان اقصر path، يعني لو انت وصلت للnode من أول step، يبقى كدا ده اقصر طريق، وهو اخد مواصلة واحدة بس، لو وصلت بعد ٤ خطوات يبقى أقصر طريق هو ال fourth degree.

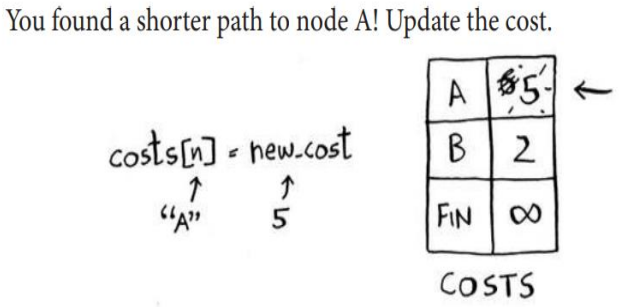
ازاي الgraph بنمثله في لغة البرمجة؟
اولا هو بيكون متكون من hash tables/dictionaries ، فمثلا الkeys هتعبر عن الnodes و الvalues هتعبر عن الconnections، يعني الkey في المثال بتاعنا هيعتبر المكان اللي احنا فيه، و الvalues هي الأماكن اللي نقدر نوصلها منه باستخدام المواصلات المختلفة.
ثانيا، بندخل كل الvalues دي في queue ، بحيث تبقى كل حاجة مترتبة حسب درجة القرب و الأهمية، فمثلًا انت في أول خطوة هتضيف كل الاحتمالات بتاعتها فال queue، وتشوف لو الdestination بتاعتك موصلتلهاش يبقى هتاخد step تانية وتضيف كل الاحتمالات بتاعتها فال queue، و هكذا، بحيث انك متطلعش من الfirst degree غير لما تكون تأكدت انك موصلتش للجهة بتاعتك فيها، و بعدين تدخل بالترتيب على الاحتمالات الي فال second degree و هكذا.
و احنا بنستخدم الqueue عشان ننظم الترتيب بنظام first in first out، بحيث انك كل ما تضيف احتمالات هتتحط فالاخر وهتفضل تلف فيها خطوة خطوة بالترتيب، الأقرب ثم الأبعد.
في الاخر وأنا عارف ان شرح حاجة زي الgraphs من غير صور شيء صعب جدًا، هو بس اقترح حل لمشكلة انك لو عندك 2 nodes متصلين ببعض فانت ممكن تقع في infinite loop، تاخد خطوة من A توصل للاحتمال الوحيد وهو B، وبعدها تاخد خطوة تانية توصل للاحتمال الوحيد وهو A، والتالتة توديك B وهكذا، الحل انك تضيف كل احتمال انت مشيت فيه قبل كدا و استكشفته في list فاضية،

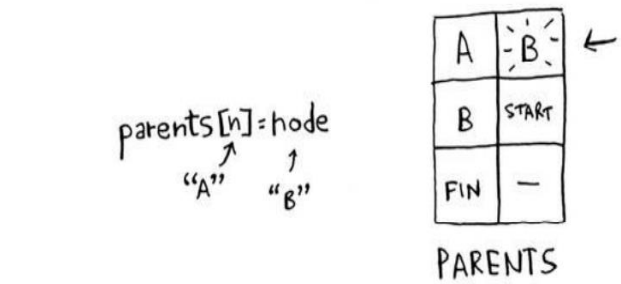
بحيث انك متضطرش تمشي في طريق مرتين، ف لو الاحتمال انت اولردي مشيت فيه و موجود جوا الليست دي هتعمله skip.

الكود النهائي هيكون بالشكل الاتي:

```
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)
```

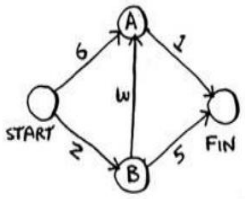
في النهاية توصلنا لان سرعة الألجورذم ده -بما انه بسيط و بياخد كل الاحتمالات المتاحة لحد ميوصل للهدف بتاعة- هي(O(V+E، حيث ان V تعبر عن عدد ال nodes، و E تعبر عن عدد ال edges.



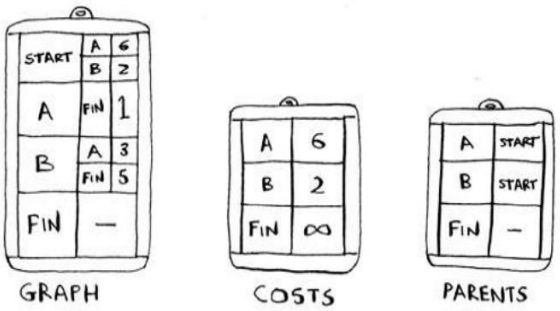


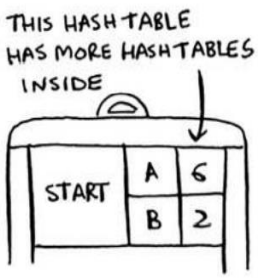
Chapter 7

في الشابتر ده الكاتب اتكلم عن ألجورذم اسمه مش هحاول أنطقه الصراحة "Dijkstra's" وأنت وضميرك في القراية، المهم أن الألجورذم ده ضاف حبة تعقيد شوية على طريقة عمل الbreadth-first search، بحيث أنه بيستعمل في إيجاد أقصر طريق في weighted graph.
يعني ايه weighted graph، يعني ببساطة ان كلedge عليها قيمة معينة، بتعبر عن الcost اللي بتدفعه عشان توصل لل node اللي في آخر ال edge.



ممكن تعتبر ال weight ده عبارة عن الزمن اللي بتقطعه عشان توصل من مكان لمكان، أو السعر بتاع الitem دي مثلًا.
الفكرة أن الألجورذم ده بيساعدك توصل لاقصر طريق، او تعمل minimize لقيمة معينة، مثلًا ال cost عشان توصل من node للتانية، أو تعمل minimize لل distance وهكذا.
طريقة عمله للأسف مش بسيطة وحتى من شرح الكاتب مفهمتش حاجة غير لما هو بدأ يمشي فالكود خطوة خطوة ويشرحه بالصور و حط القيم بتاعة كل متغير في الكود، هحط الصور دي للاسف كاملة لأني مظنش إني أقدر أشرح الموضوع بطريقة أحسن من كدا بالكتابة أبدًا.
بس هو حاجة لازم تتعرف الأول قبل ما نشوف الكود والخطوات بتاعته، انت بتحتاج ٣ hash tables:





١- للجراف كله، بيتكون من nodes كلها وكل النودز دي مرتبطة ب nodes ايه تاني، و ال weights اللي بتربطهم ببعض.

٢- لل costs او ال weights من نقطة البداية لكل ال nodes التانية، وده هنستخدمه في إننا نسجل اقل cost عشان نوصل لل node دي.

٣- لل parents، و هو هنستخدمه عشان نقول انا وصلت من هنا لهنا ولقيت ان ده أقل cost فعشان أحفظ أنا بدأت منين و روحت فين بأقل قيمة، لازم يكون عندنا الجدول ده عشان نعدّل فيه.

بالنسبة لكود الألجورذم:

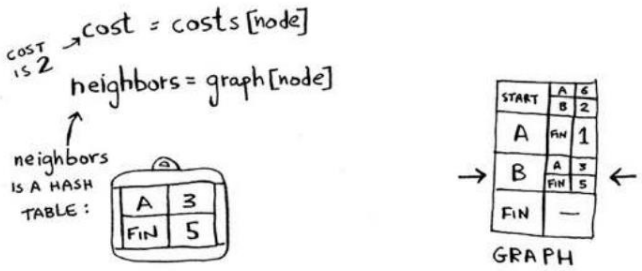
```
node = find_lowest_cost_node(costs)   # Find the lowest-cost node that you haven't processed yet.
while node is not None:               # If you've processed all the nodes, this while loop is done.
    cost = costs[node]
    neighbors = graph[node]
    for n in neighbors.keys():        # Go through all the neighbors of this node.
        new_cost = cost + neighbors[n]   # If it's cheaper to get to this neighbor
        if costs[n] > new_cost:          # by going through this node ...
            costs[n] = new_cost          # ... update the cost for this node.
            parents[n] = node            # This node becomes the new parent for this neighbor.
    processed.append(node)               # Mark the node as processed.
    node = find_lowest_cost_node(costs)  # Find the next node to process, and loop.
```

```
def find_lowest_cost_node(costs):
    lowest_cost = float("inf")
    lowest_cost_node = None
    for node in costs:                   # Go through each node.
        cost = costs[node]
        if cost < lowest_cost and node not in processed:   # If it's the lowest cost so far and hasn't been processed yet ...
            lowest_cost = cost           # ... set it as the new lowest-cost node.
            lowest_cost_node = node
    return lowest_cost_node
```

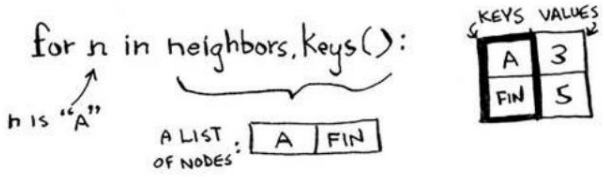
وهنا الخطوات الي بياخدها الألجورذم بناءً على الجراف الي فالأول:

Find the node with the lowest cost.

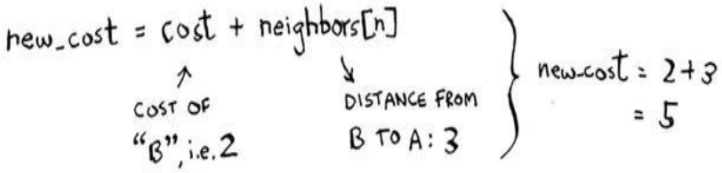
Get the cost and neighbors of that node.



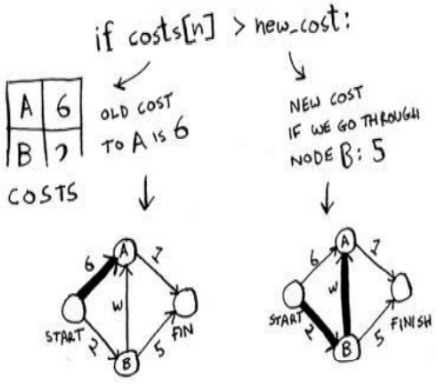
Loop through the neighbors.



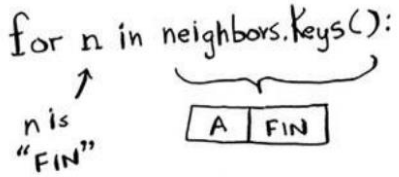
Each node has a cost. The cost is how long it takes to get to that node from the start. Here, you're calculating how long it would take to get to node A if you went Start > node B > node A, instead of Start > node A.



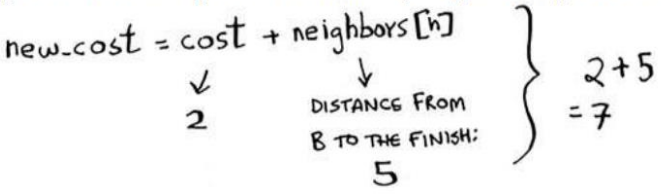
Let's compare those costs.



Ok, you're back at the top of the loop. The next neighbor `for` is the Finish node.

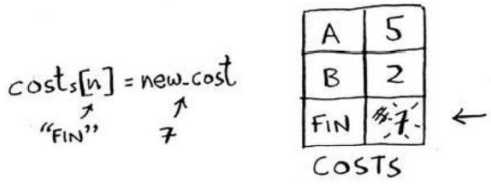


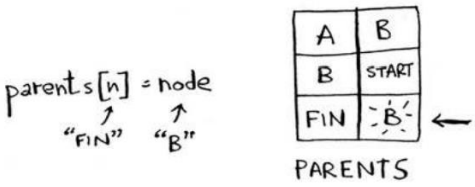
How long does it take to get to the finish if you go through node B?



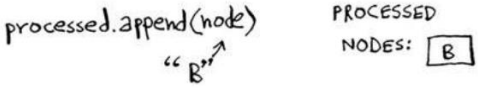
It takes 7 minutes. The previous cost was infinity minutes, and 7 minutes is less than that.

Set the new cost and the new parent for the Finish node.

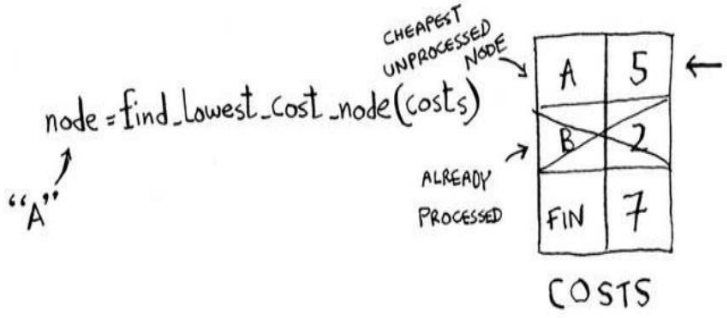




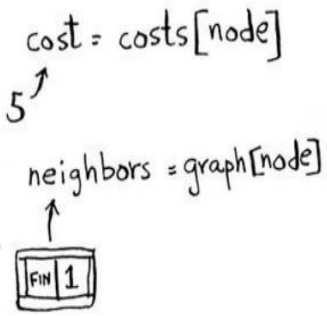
Ok, you updated the costs for all the neighbors of node B. Mark it as processed.



Find the next node to process.

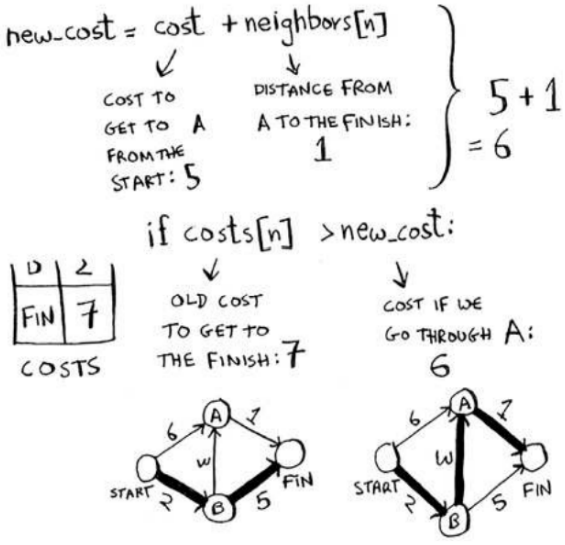


Get the cost and neighbors for node A.

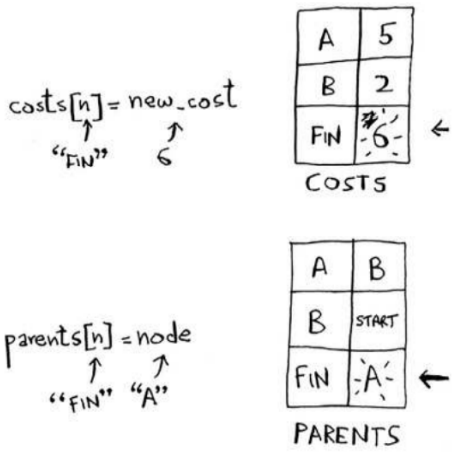


Node A only has one neighbor: the Finish node.

Currently it takes 7 minutes to get to the Finish node. How long would it take to get there if you went through node A?



It's faster to get to Finish from node A! Let's update the cost and parent.



من عيوب الألجورذم بقى انه مبيشتغلش غير لو كل ال weights تكون موجبة، مش بيشتغل لو في weight سالب، في حالة ان في weights سالبة بنستخدم ألجورذم تاني اسمه Bellman-Ford، الكاتب متكلمش عنه بس ممكن أحط لينك شرح ليه هنا للمهتم. Bellman-Ford

Chapter 8

في الشابتر ده الكاتب خلص كلام عن الgraphs و بدأ يتكلم عن الgreedy algorithms، و هو نوع بسيط جدا من الألجورذمز اللي بنستخدمه عشان نوصل لlocal solution، أو رياضيًا إننا نعمل estimation للناتج اللي عاوزينه، وبنستخدم الطريقة دي عشان نحل المشكلات اللي ملهاش حل exact فبنوصل لحل تقريبي منطقي.
زي ما قولت انه بسيط جدًا فهو ممكن أي مشكلة تحلها بيه بسهولة وبساطة، زي مثلًا عاوز تعرف أقصر طريق تاخده عشان توصل لنقطة ما، فهتبدأ في نقطة، وتمشي لأقرب نقطة ليها، ثم أقصر نقطة ليها، وهكذا، لحد متوصل للنهاية.
ايه أنواع المشاكل دي أو أشكالها؟، هو يقدر يتحل بيه أي مشكلة زي ما قولنا من نوع NP-complete اللي هي زي مشكلة الsalesman وكدا.
من المشاكل دي برضو إن يبقى عندك set فيها مجموعة عناصر وset تانية وتالتة وهكذا، وأنت بتحاول توصل لcombination معينة بينهم، تحت شروط وقواعد محددة، زي مثلًا إنك عندك أكونتات أفراد، كل فرد فيهم ليه set of preferences، وأنت بتحاول تقسم بين الأفراد دول حسب اللي عنده preferences أكتر مع التاني، عشان تحل مشكلة زي دي محتاج تحسب مدى التشابه بين كل أكونت والتاني، وده هيحتاج إنك تحسب رقم بيعبر عن التشابه أو الاختلاف بين كل فرد والتاني، وده هيخلي عدد الاحتمالات مهول وكبير جدًا، بزيادة عدد العناصر اللي عندك.
حل لمشكلة زي دي، إنك تبني زي hierarchy، تمسك شخص "local" وتشوف مين أشبه شخص بيه، خلاص وصلتله؟ تروح تشوف مين أشبه شخص بالشخص ده، وهكذا.
بحيث إنك تبني زي بناء أو خطوات، كل خطوة بتتبع constrain معين -اللي في الحالة دي هو الشبه، أكتر حد شبه بالتاني-، و آخر شخص ده بيبقى فالأغلب فعلًا أكتر شخص مش شبه اللي بدأت بيه، و قولنا في الأغلب عشان أنت ممكن لو بدأت بداية تانية يطلع الناتج شخص تاني قريب منه او حواليه.
الألجورذم بسيط وسهل بيعتمد إنه يختار أضمن اختيار أو أعلى اختيار في العائد -على حسب الشروط بتاعة المشكلة-، وده السبب انه بيخليه estimator مش بيطلع القيمة الفعلية، يعني بدل ما يحسب العلاقة بين كل عنصر والتاني، لا هو بيمسك عنصر ويختار أقرب عنصر ليه بيحقق العلاقة و بعدها ياخد أقرب عنصر للعنصر ده وهكذا.

الخلاصة بقى:

- مش بتستخدم الgreedy approach ده غير لو مشكلتك كبيرة وإنك تحسب حلها هياخد وقت كبير جدًا، وفي المشاكل اللي مش محتاج الexact solution وإنك ممكن تقبل بestimate بس.

- لو عندك مشكلة NP-complete يبقى أحسن حل إنك تستخدم greedy approach

و مشاكل الNP-complete ممكن تعرفها من الآتي:

- لو في ألجورذم كان شغال حلو وسريع معاك عادي، ولما زودت الداتا لقيت السرعة قلت بشكل كبير جدًا وexponentially.

- لو لقيت إنك محتاج عشان تحل المشكلة بتاعتك تحسب كل الاحتمالات الممكن حدوثها، all possible values of X، أيًا كانت X دي ايه.

- لو لقيت إن عندك sequence أو set من القيم و بتحاول توصل لعلاقة بين القيم دي، أو بتحاول تجاوب عن سؤال معين أو "constraints" أنت بتحاول تطبقها على القيم.

Chapter 9

في شابتر ٩ الكاتب بيتكلم عن ال dynamic programming وهو موضوع تقيل جدًا وملهوش ملكة يعتبر، كله واحد ممكن يحله بطريقة وملهوش خطوات ثابتة يتمشى عليها
هو طريقة تفكير لحل مشاكل معينة، والمشاكل دي أغلبها بتبقى مشاكل تعتمد على إنك بتحاول تعمل constrained optimization، مثلًا قيمة بتحاول تخليها max/min تحت شرط معين أو ظرف معين، والطريقة دي بتخليك تقدر تكتب انت algorithms مخصصة لمشكلتك دي، عن طريقة تجزيء المشكلة دي لمشاكل أصغر تقدر تحلها بسهولة ومن اللي وصلتله توصل لحل المشكلة الأصلية.

طب نتكلم عن الطريقة دي مبنية على ايه، أو أساسياتها ايه:

١- بتحول المشكلة لجدول grid.

٢- قيم الcells هي القيم اللي أنت بتحاول تعمل لها optimization، وتقدر تعتبرها هي دي المشاكل الصغيرة أو الsub-problems-.

٣- المحاور بتاعة الجدول بتبقى عبارة عن المتغير بتاعك، وده بيختلف من تطبيق للتاني.

٤- تحل الجدول ده و تملأ القيم وتحاول تشوف طريقة تكتب بيها الخطوات دي في loop / if conditions.

وبسبب آخر خطوة دي قولت إنها طريقة تفكير مش algorithm، لإنك أنت اللي بتكتب الalgorithm على حسب أنت عاوز ايه بظبط وجدولك بيتحل ازاي.
مثال بقى عشان الموضوع بقى ضبابي أنا عارف، مثلًا عندك يا سيدي كلمة مكتوبة غلط، وبتحاول توصل للكلمة الصح اللي كنت قاصدها.
الكاتب قالك انا هعمل الجدول بين الكلمة الغلط المكتوبة وبين الكلمات التانية القريبة منها، وأشوف عدد الحروف المتشابهة بين كل كلمة والتانية واختار أعلى قيمة على إن هي دي الكلمة الصح.

- خلى المحاور عبارة عن حروف الكلمة بتاعتك وحروف الكلمة التانية على المحور التاني - for word in close_words - حاجة زي كدا.

- خلى قيم الcells هو عدد الحروف المتكررة بين الكلمتين، أو عدد الحروف المتكررة ورا بعض.

فضل يحسب بقى لكل جدول القيم اللي فيه وفي الاخر وصل لalgorithm من سطرين، وبيكون الحل معتمد على طريقة الgrid، فمثلًا:

```
If word_a[i] == word_b[j]:
   Cell[i][j]= cell[i+1][j+1] + 1
```

إلخ..
الفكرة مش في الكود هنا، الفكرة انه كتبه على حسب الطريقة اللي ملأ بيها الجدول في الأساس.
في الآخر الكاتب بيعرض بعض استخدامات الطريقة دي في حل مشاكل الcommon sub-sequence، و هي نفس نوع مشكلة الكلمات اليي شبه بعض، اعتبرها مشكلة similarity وبتحاول تلاقي الpattern وسط الحاجات عشان تقارن بينها.

زي مين بقى بيستخدم الdynamic programming عد معايا:

أولًا الناس بتوع البايو وهم بيقارنوا الDNA لحيوانات مختلفة او لأمراض مختلفة.

ثانيًا حاجة زي git أو google doc لما بيوريك أنت غيرت أو عدلت أد ايه من الversion القديمة للفايل ده، ويقولك غيرت ايه بالظبط.

ثالثًا ألجورذم معروف اسمه levenshtein distance ودي طريقة بتعرف بيها الsimilarity بين نص والتاني، وبيستخدم في الspell-checking والcopyrighted material، ومش بعيد يكون بيستخدم في الplagiarism عشان يحدد هل البحث العلمي ده مسروق ولا لأ.

رابعًا مش مجرد مشاكل الsimilarity، عندك برضو doc/word بيستخدموها عشان يعملوا wrap للكلام وينظموه بحيث إن الكلمات تبدأ و تنتهي في السطر بنفس المسافة، فهنا أنت بتحاول تعمل optimize لأبعاد النص فالصفحة -غالبا يعني-
الخلاصة إن الdynamic programming موضوع تقيل جدًا، وأنا في الاول فكرته هيبقى زي الrecursion بس طلع معقد أكتر بسبب موضوع الcells-grid، وإنك لازم تحل المشكلة بإيدك وتحاول تترجم الخطوات بتاعتك لcode وsteps للألجورذم.

Chapter 10

نتكلم بقى في الشابتر المقرب لقلبي وهو متعلق بالmachine learning بشكل عام وبيدي لمحة عن تفاصيله باستخدام k-nearest neighbors.

ايه هو ال knn؟
هو موديل بسيط تقدر تعمل بيه classification أو regression.

طب ايه دول؟
الclassification هو ببساطة انك تحدد نوع أو علاقة معينة، مثلا ده كلب ولا قطة، نوع الفاكهة تفاح ولا مانجا، وهكذا.
أما الregression ف هو انك تقول قيمة عددية continuous، يعني في ال classification فانت عندك classes محددة بتختار واحد منهم عشان تطلعه كـ output، أما هنا فانت بتطلع قيمة عددية ممكن من صفر لمالا نهاية او من سالب مالا نهاية لمالا نهاية، زي مثلًا سعر سلعة، أو درجة حرارة.

طب ليه knn؟
لانه بسيط جدًا جدوا، بيعتمد على انه بيشوف الinput ده موقعه فين بالنسبة لكل الداتا اللي هو شافها قبل كدا، كأنه بيعمل plot للداتا اللي داخلة دي في input space ويقارنها بكل الداتا التانية، ويشوف مثلًا انت محدد قيمة k بكام -فلنقل ٥- ، ف هو ساعتها هيروح يشوف أقرب ٥ قيم للنقطة الجديدة، و يشوف قيمتها/نوعها ايه، و على الأساس ده بيحدد قيمة/نوع الداتا الجديدة.
بس فيه نقطتين لازم تفهمهم عشان هم دول اللي قائم عليهم الموديل:
١- أهمية الfeatures، و هي البيانات اللي داخلة، يعني بتقارن فاكهتين يبقى من الصفات اللي بتحددهم اللون والشكل، متجيش تدخل انت مثلًا تاريخ الحصاد بتاعها.
لو بتعمل recommendation مثلًا لأفلام، فبتشوف الحاجات اللي عجبت العميل أو اللي عملها ريت عالي، او الpreferences بتاعته أو الفئة العمرية، لكن من الحاجات اللي ممكن تبقى مش مهمة هي شيفت العمل بتاعه صباحي ولا مسائي، وهكذا.

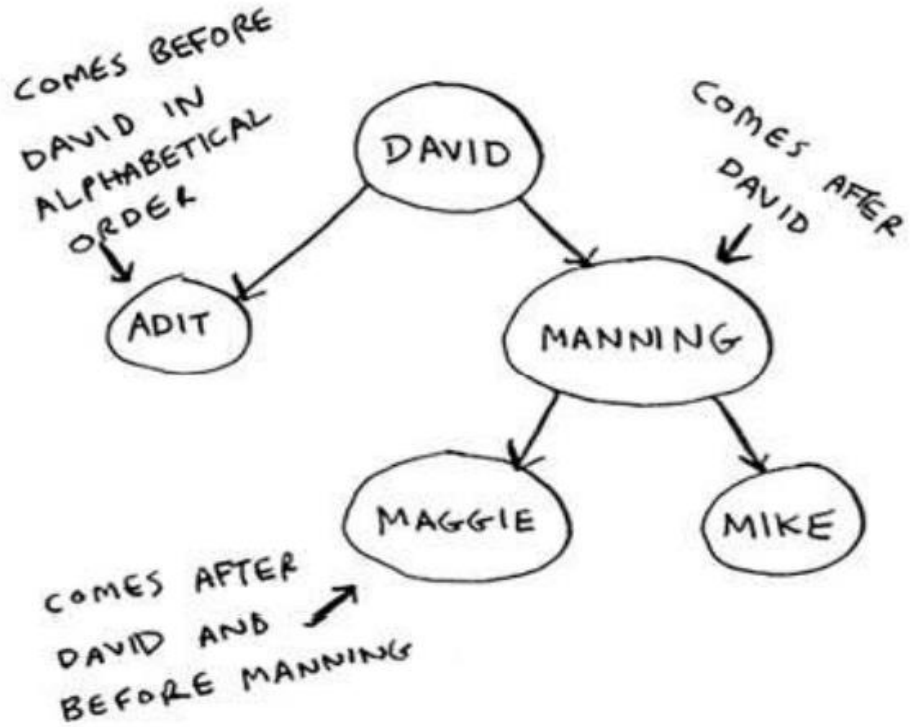
٢- على أي أساس بتحسب القرب والبعد والمسافة؟
أبسط اجابة ممكن تكون بقانون فيثاغورث جذر و تحتيه الفرق تربيع بين صفة ونفسها في النقطة التانية، بس دي ممكن لسهولتها تطلع إجابتين شبه بعض بالظبط مع انهم مختلفين تمامًا لو جذر (١-٢)+(٠-١) هيديك قيمة ولتكن x، و برضو جذر (٠-١)+(٣-٤) هيديك نفس القيمة، مع الأخذ فالاعتبار ان الfeatures مرتبة.

عشان كدا فيه أنواع مختلفة من الdistance metrics زي الcosine distance، ودي بتستخدم في حساب الزاوية بين نقطتين وبكده حلت العيب وعرفت ان اه القيمتين هما هما بس الاتجاه معكوس.
و ده GIF بيوضح طريقة العمل بالخطوات.
بعد كدا الكاتب ادى أمثلة لاستخدام الماشين ليرننج بشكل عام زي الface recognition أو الOCR، واتكلم عن أهمية الfeatures، و ادى مثال جميل لمشكلة ملهاش حل لحد دلوقتي وهي التنبأ بأسعار البورصة، وشرح الموضوع بشكل جميل ان بالنسبالك ايه اللي ممكن يأثر على سعر سهم ما في المستقبل؟ هل هو سعره قبلها بيوم؟ شهر؟ اسبوع؟ حال البلد المادي؟ حال الشركة المادية؟ استقالات فالشركة مثلا؟ عمل ارهابي حصل مثلًا؟، عدد المتغيرات اللي ممكن تأثر على التنبآت في المستقبل كبير جدًا، ولذلك مشكلة زي دي حلها بيبقى مش سليم ونتيجته مش مرضية، وبكده عرض بشكل جميل تأثير الfeatures اللي بتستخدمها على كفاءة الmodel بتاعك.

Chapter 11

السلام عليكم، في آخر شابتر معانا شابتر ١١ في الكتاب الرائع ده، الكاتب بيتكلم عن ١٠ ألجورذمات يعتبروا advanced وهو مشرحهمش ولا اتكلم عنهم وسايبهم ليك لو حابب تتعمق وتتعلم أكتر في حاجة معينة فيهم، على حسب نطاق عملك واهتماماتك.

١- الbinary search trees:
وهو نوع من الtrees اللي بيستخدم بدل الarrays في التعامل مع الداتا (تخزين - بحث - مسح)، وهو أسرع من الarray في التخزين والمسح حيث انه O(log n) من حيث السرعة، وهي فكرتها بسيطة حيث ان كل node في الtree بتحتوي قيمة والchildren بتوعها بيكونوا اتنين قيمة قبل الparent ودي على الشمال، وقيمة بعد الparent ودي على اليمين.



وده بيوفر إنك كل شوية مش محتاج تعيد ترتيب الarray لما تغير فالقيم بتاعتها.
والألجورذم ده بيستخدم غالبًا في حفظ الداتا بداخل الdatabase، ويوجد منه العديد من الvariations لكل منها مميزات وعيوب.

٢- الInverted indexes:
وهو ببساطة استخدام الhash tabels عشان تشير لindex في data structure تانية.

مثلًا عندك dictionary الkey بتعبر عن كلمة، والvalue بتعبر عن رقم الصفحات اللي ظهرت فيها الكلمة، انت لو دخلت الكلمة بتاعتك هيطلعلك ال index اللي بيقع فيه الvalue الحقيقية اللي انت عاوزها، فهتروح على الكتاب/الdoc اللي بتدور جواه، وهتدخل الindex ده، وساعتها هيطلعلك اللي انت بتدور عليه.
هو طريقة سهلة وبسيطة للبحث، و تعتبر استخدام من استخدامات الhashing tabels عوضًا عن انها algorithm مختلف.

٣- الFourier transform:
وهو أصعب حاجة ممكن تقابل طالب في هندسة، بس حاجة مفيدة جدًا في مختلف المجالات، فممكن استخدامها لتحليل أي signal للمكونات بتاعتها، مثلًا لو أغنية ف هيطلع كل الfrequencies اللي بتشكل الأغنية دي، ويدي لكل frequency فيهم weight يعبر عن أهميتها، وده طبعًا بيستخدم في عمليات الcompression زي الmp3 وصيغ الصور المختلفة، بحيث انهم بياخدوا الfrequencies الأعلى في الweigths ويهملوا الباقي لانه مش مهم في اعادة تكوين الملف الأصلي.
طبعا برضو بيستخدم في حاجات مختلفة زي الصور و الsequential data وأي حاجة ممكن تتجزأ لfrequencies أصغر لما يتجمعوا يطلعوا الداتا الأصلية.

٤- الparallel algorithms:
و ده شبه الdynamic programming كده، طريقة تفكير ومجال علمي واسع، باختصار بيتكلم عن كيفية تجزئة الalgorithms على أجهزة مختلفة، بحيث ان الوقت المستغرق من ألجورذم شغال على جهاز واحد يقل لما نجزأه على جهازين، بحيث ان كل جهاز يعمل نص الشغل و الاتين في الاخر لما يخلصوا يتجمعوا أو يتعمل لشغلهم merge.

ليه ده concept صعب؟
بسبب انك محتاج تدرس كويس تأثير العوامل المختلفة، زي لو عندك array من ألف عنصر، هتقسم العناصر بالتساوي على الأجهزة؟ يعني لو معاك جهازين هتدي لكل واحد ٥٠٠؟، طب والوقت بتاع الmerge هيبقى أد ايه مثلا؟
برضو لو عندك كذا task هتقسم لكل جهاز تاسك معين؟ طب افرض فيه تاسك وقته أقل من التاني بكتير، هتبقى كدا شغال بجهاز اغلب الوقت والجهاز التاني واقف مش بيشتغل.
لازم كل الحاجات دي تتحسب كويس، و عشان كدا هو field of study لأي حد مهتم بال scalability and performance.

٥- الMapReduce:
و ده ألجورذم مهم جدًا فالتعامل مع الbig data وبيستخدم في الdatabases اللي عندها ملايين العناصر، وبيستخدم في الdistributed systems ويعتبر جزء من عائلة الdistributed algorithms ، واللي بتساعدك انك تقوم بعمليات كتيرة جدا ف وقت قليل جدًا.
وهو بيعتمد على عمليتين:
أولًا الmapping، وهي انك تعمل map من حاجة لحاجة تانية، مثلا بتعمل ماب من array لfunction، يعني بتعدي على كل قيمة فالarray وتعمل عليها عملية حسابية موجودة في الfunction، فبكده بتدخل array وبتطلع array.
ثانيًا الreduction، وهو انك بتقلص حاجة كبيرة لحاجة صغيرة، مثلًا من array لقيمة واحدة، كإنك بتعمل hashing ليها مثلًا، فبتاخد array من قيم مختلفة وترمزلها بقيمة واحدة فقط.
وهو بيستخدم الطريقتين دول في عمل عمليات كتيرة جدًا على أجهزة مختلفة في وقت قليل، زي الqueries في big data databases.

٦- الbloom filters & hyperloglog:
و دول ألجورذمز بيستخدموا في البحث جوا sets، بس مش مجرد set بسيطة، لا بتستخدم للبحث ف set جواها ملايين العناصر، مثلًا أسماء الusernames في الفيسبوك، أو الunique search words في أي search engine.
الفكرة انهم يقدروا ببساطة يحتفظوا بالset دي في شكل hash tabel وساعتها لما يحبوا يشوفوا هل الاسم موجود عندنا؟ ساعتها ده هيبقى وقت البحث(O(1 ، بس المشكلة هنا مش في السرعة، الفكرة في الكمية، عشان تحفظ hash tabel جواه ملايين العناصر يبقى محتاج مساحة مهولة، وده اللي مش دايما بيكون متاح، انت بتحاول تحل مشكلة المساحة هنا.
طب ايه الحل؟
بكل بساطة انك تستخدم ألجورذم يديك probability تقريبية، هل العنصر ده موجود عندي في ال set ولا لأ؟ و ده اللي بيعمله الbloom filter، من عيوبه انه ممكن يديك False positive يعني يقولك هو اه عندنا، مع انه مش موجود، و من مزاياه انه عمره مهيطلع false negative، يعني لو قالك مش عندنا يبقى ١٠٠٪ مش موجود.
أما ال hyperloglog ده فهو بيعمل زي البلوم بظبط لكن بيشتغل على subset من الset بتاعتك وبيطلع الناتج بدقة أكتر.
والاتنين دول بتستخدمهم في حالة انك مش متضر لو اشتغلت بprobability أو approximation، عشان كده اسمهم probability algorithms.

٧- الSHA algorithms:

وده يعتبر hashing function بتستخدم لتحويل داتا إلى hash معين، كانك بتعمل encryption للداتا دي بطريقة معينة.
بتقدر طبعًا تستخدم الألجورذم ده عشان تقارن ملفين شبه بعض ولا لأ، هل حصل فيهم تغيير؟، مثلًا جوجل بتستخدمه عشان تعرف هل الباسورد اللي دخلته ده صح ولا غلط؟ مش هتقارن الباسورد الحقيقي باللي موجود عندك، لا بتقارن ال SHA بتاعك بال SHA المسجل عندهم.
و من مميزات الSHA انه مش بيتأثر ببساطة، يعني لو غيرت حرف واحد من الكلمة اللي مدخلها، مش معنى كدا ان الSHA بتاعها هيبقى شبه الكلمة الأصلية، لا إطلاقًا كل داتا ليها SHA مختلف، بس المبرمجين استخدموا SImHash وده فيه الخاصية بالعكس، ان لو غيرت حرف واحد الSHA بتاعها هيبقى مقارب جدًا للكلمة الأصلية، وده بيستخدموه زي كاشف لحقوق الملكية وال plagerism.

٨- الDiffie-Hellman key exchange:
و ده ألجورذم بيستخدم في الencryption، وحل مشكلة ازاي أحافظ على الأمان بتاع المستخدمين، هو مدخلش في تفاصيل أوي، بس الخلاصة انه انت بتحتاج مفتاح عشان تفك أي شفرة، و بيكون الطرفين اللي بيتناقلوا المعلومات معاهم المفتاح ده عشان يحلوا الشفرة ويرجعوا المعلومات لأصلها، فمثلًا لو المفتاح ده عبارة عن ان كل حرف مكتوب بالbinary، فعشان تفك الشفرة هترجع الbinary لحروف وشكرًا.
فكرة الألجورذم ده بقى عشان يحل مشكلة ان ممكن المفتاح ده يتسرب وبيكون لازم تغثره كل فترة و كدا، انه استخدم مفتاحين، مفتاح public وده بتستخدمه عشان تعمل encryption، ومفتاح private وده بتستخدمه عشان تعمل decryption، المفتاح الpublic بيكون متاح لكل الusers، أما المفتاح الprivate فده مش موجود غير مع الطرفين اللي بيتناقلوا المعلومات بس.

٩- الLinear programming:
وتاني ده لا يعتبر ألجورذم، بس هو طريقة تفكير منهجية، و دي الطريقة اللي بنفكر بيها لما بنكون عاوزين نعمل maximization لحاجة معينة في وجود constraints، ودي الطريقة اللي اتبعناها في الgraphs عشان نوصل لأقصر طريق أو أعلى ربح ممكن ناخده.
وأضاف الكاتب ان الlinear programming بتستخدم الsimplex algorithm ، وهو ألجورذم معقد، لكن مهم جدًا لأي حد مهتم بمشاكل ال optimization.

و بكده نكون ختمنا كتاب **Grokking Algorithms** بحمد الله