Assignment 4: ADTs and Recursion

作业 4：ADT 和递归

Huge props for Keith Schwarz for devising all of these problems, and kudos to Julie Zelenski, Marty Stepp, and Jerry Cain for assisting Keith with their construction. Special thanks to Nick Bowman and Eli Echt-Wilson for providing historical election data.

基思-施瓦茨（Keith Schwarz）设计了所有这些问题，朱莉-泽伦斯基（Julie Zelenski）、马蒂-斯蒂普（Marty Stepp）和杰里-凯恩（Jerry Cain）协助基思构建了这些问题，在此向他们致以崇高的敬意。特别感谢 Nick Bowman 和 Eli Echt-Wilson 提供历史选举数据。

Motivation 动机

Recursion is an extremely powerful problem-solving tool with tons of practical applications. This assignment consists of three real-world recursion problems, each of which is interesting in its own right. By the time you're done, you'll have a much deeper appreciation both for recursive problem solving and the wide range of problem domains where recursion can be applied.

递归是一种非常强大的解决问题的工具，有大量的实际应用。本作业包括三个现实世界中的递归问题，每个问题本身都很有趣。完成作业后，你将对递归问题的解决和递归的广泛应用有更深刻的认识。

This assignment explores a lot of different concepts in recursion. The assignment, however, intentionally leaves the gaming domain behind and asks that you apply recursion and recursive backtracking to scientifically interesting problems that come up in practice. It also has the lovely side effect of revisiting a good number of the containers-Set, Map, vector, etc.-that you learned about during the first two weeks of the quarter. tl; dr : It's great preparation for your first midterm.

本作业探讨了递归中的许多不同概念。不过，这项作业有意抛开了游戏领域，而要求你将递归和递归回溯应用到实践中出现的有趣的科学问题上。这项作业还有一个很好的副作用，那就是重温了本季度前两周学习的大量容器--集合、映射、向量等。 tl; dr ： 这是为第一次期中考试做的很好的准备。

If you start typing out code without a clear sense of how your recursive solution will work, chances are you'll end up going down the wrong path. Before you begin, try thinking about the recursive structure of problem solution you're pursuing. Does anything fall into one of the nice categories we've seen before (listing subsets, permutations, combinations, etc.)? When using backtracking, what choices can you make at each step? Talking through these questions before you start coding and making sure you have a good conceptual understanding of what it is that you need to do can save you many, many hours of coding and debugging. Note that all of your work should be placed in the

如果你不清楚递归解决方案是如何工作的，就开始敲代码，很可能最终会走错路。在开始之前，试着思考一下你所追求的问题解决方案的递归结构。是否有任何东西属于我们之前见过的好类别之一（列出子集、排列、组合等）？在使用回溯法时，每一步都可以做出哪些选择？在开始编码前讨论这些问题，并确保对需要做的事情有很好的概念性理解，可以节省很多很多小时的编码和调试时间。请注意，您的所有工作都应放在

RecursionToTheRescue.cpp file, as all of the others are there for the test framework you should rely on to exercise your code.

RecursionToTheRescue.cpp 文件，因为所有其他文件都是为测试框架准备的，你应该依靠它来练习你的代码。

Due: Wednesday, October 23rd at 11:59 pm [1]

提交时间：10 月 23rd 日（星期三）晚上 11:59 [1]

Problem 1: Doctors Without Orders

问题 1：没有医嘱的医生

You're tasked with helping the country of Recursia build out its health care system, and now it faces a crisis! No one has told the Recursian doctors which patients to see - they're Doctors without Orders! As Minister of Health, it's time to help the Recursians with their medical needs.

你的任务是帮助 Recursia 国家建立医疗保健系统，现在它面临着一场危机！没有人告诉雷丘西亚的医生该给哪些病人看病--他们是没有医嘱的医生！作为卫生部长，是时候帮助雷丘西亚人解决他们的医疗需求了。

Consider the following two record definitions designed to represent doctors and patients:

请看下面两个分别代表医生和病人的记录定义：

```
struct Doctor {
    string name;
    int hoursFree;
};
struct Patient {
    string name;
    int hoursNeeded;
};
```

Each doctor has a number of hours that they're capable of working each day, and each patient has a number of hours they need to be seen. Your task is to write a function

每个医生每天都有一定的工作时间，每个病人也有一定的就诊时间。您的任务是编写一个函数

```
bool canAllPatientsBeSeen(const Vector<Doctor>& doctors,
    const Vector<Patient>& patients,
    Map<string, Set<string>>& schedule);
```

that takes as input a list of available doctors, a list of available patients, and then returns whether it's possible to schedule all the patients so that each one is seen by some doctor for the required amount of time. If it is possible to schedule everyone, the function should fill in the final schedule parameter by associating each doctor's name with the set of the names of patients he or she can see.

该函数输入可用医生列表和可用病人列表，然后返回是否可以安排所有病人的就诊时间，以便每个病人都能在规定的时间内由某个医生接诊。如果可以安排每个人的就诊时间，函数应将每个医生的姓名与他或她可以就诊的病人姓名集合关联起来，从而填入最后的日程参数。

For example, suppose we have these doctors and patients:

例如，假设我们有这些医生和病人：

Patient Lacks: 2 Hours Needed

病人缺乏：需要 2 小时

Doctor Thomas: 10 Hours Free

托马斯医生：10 小时免费

Doctor Taussig: 8 Hours Free

陶西格医生：8 小时免费

Doctor Sacks: 8 Hours Free

麻袋医生8 小时免费

Doctor Ofri: 8 Hours Free

奥弗里医生：8 小时免费

Patient Gage: 3 Hours Needed

病人计：需要 3 小时

Patient Molaison: 4 Hours Needed

病人莫莱森：需要 4 小时

Patient Writebol: 3 Hours Needed

病人 Writebol：需要 3 个小时

Patient St. Martin: 1 Hour Needed

圣马丁病人：需要 1 小时

Patient Washkansky: 6 Hours Needed

病人 Washkansky：需要 6 个小时

Patient Sandoval: 8 Hours Needed

病人桑多瓦尔需要 8 小时

Patient Giese: 6 Hours Needed

病人 Giese：需要 6 个小时

In this case, everyone can be seen:

在这种情况下，每个人都能被看到：

Doctor Thomas (10 hours free) sees Patients Molaison, Gage, and Writebol (10 hours total)

托马斯医生（10 小时免费）为莫莱森、盖奇和 Writebol 病人看病（共 10 小时）

Doctor Taussig (8 hours free) sees Patients Lacks and Washkansky (8 hours total)

陶西格医生（免费 8 小时）为拉克斯和瓦什坎斯基病人看病（共 8 小时）

Doctor Sacks (8 hours free) sees Patients Giese and St. Martin (7 hours total)

萨克斯医生（免费 8 小时）为吉斯和圣马丁病人看病（共 7 小时）

Doctor Ofri (8 hours free) sees Patient Sandoval (8 hours total)

奥弗里医生（免费 8 小时）为桑多瓦尔病人看病（共 8 小时）

However, minor changes to the patient requirements can completely invalidate the result. If, for example, Patient Lacks needed three hours instead of just two, there wouldn't be a way to schedule all the patients such that all could be seen. On the other hand, if Patient Washkansky needed seven hours instead of six, there'd still be a way to schedule everyone. (Do you see how?)

然而，病人要求的微小变化就会使结果完全失效。例如，如果病人拉克斯需要三个小时，而不是只需要两个小时，那么就无法安排所有病人都能就诊。另一方面，如果病人 Washkansky 需要 7 个小时而不是 6 个小时，还是有办法安排所有人的时间。(你明白了吗？)

This problem is all about recursive backtracking. Think about what decision you might make at each point in time. How do you commit to a decision and then rewind it doesn't work out? As always, feel free to introduce as many helper functions as you'd like. You may even want to make the primary function a wrapper around some other recursive function.

这个问题就是递归回溯。想想你在每个时间点可能会做出什么决定。你如何做出一个决定，然后在不成功的情况下进行回退？和往常一样，你可以随意引入更多的辅助函数。你甚至可以将主函数作为其他递归函数的包装。

Some notes on this problem:

关于这个问题的一些说明：

You can assume that schedule is empty when the function is called.

可以假设调用该函数时 schedule 为空。

If your function returns false, the final contents of schedule don't matter (though we suspect your code will probably leave it blank).

如果函数返回 false，则 schedule 的最终内容并不重要（不过我们猜测你的代码可能会将其留空）。

Although the parameters to this function are passed by const reference, you're free to make extra copies of the arguments or to set up whatever auxiliary data structures you'd like in the course of solving this problem.

虽然这个函数的参数是通过常量引用传递的，但在解决这个问题的过程中，你可以自由地复制额外的参数或设置任何你想要的辅助数据结构。

You can assume no two doctors have the same name and no two patients have the same name.

你可以假设没有两个医生的名字相同，也没有两个病人的名字相同。

You may find it easier to solve this problem first by simply getting the return value right and ignoring the schedule parameter. Once you're sure your code is always producing the correct return value, update it so you fill in schedule. Doing so shouldn't require too much code, and it is much easier to add this in at the end than it is to debug the whole thing all at once.

您可能会发现，首先只需获得正确的返回值而忽略计划参数，就能更容易地解决这个问题。一旦你确定代码总是能产生正确的返回值，就可以更新代码，填入时间表。这样做应该不需要太多代码，而且在最后添加这个参数比一次性调试整个程序要容易得多。

If there's a doctor who doesn't end up seeing any patients, you can either include the doctor's name as a key in schedule (associated with an empty set of patients) or leave the doctor out entirely, whichever you'd prefer.

如果有医生最终没有为任何病人看病，你可以将该医生的名字作为日程表的关键字（与一组空病人相关），或者将该医生完全排除在外，任选其一。

Problem 2: Disaster Preparation

问题 2：备灾

Disasters—natural and unnatural—are inevitable, and cities need to be prepared to respond to them when they occur.

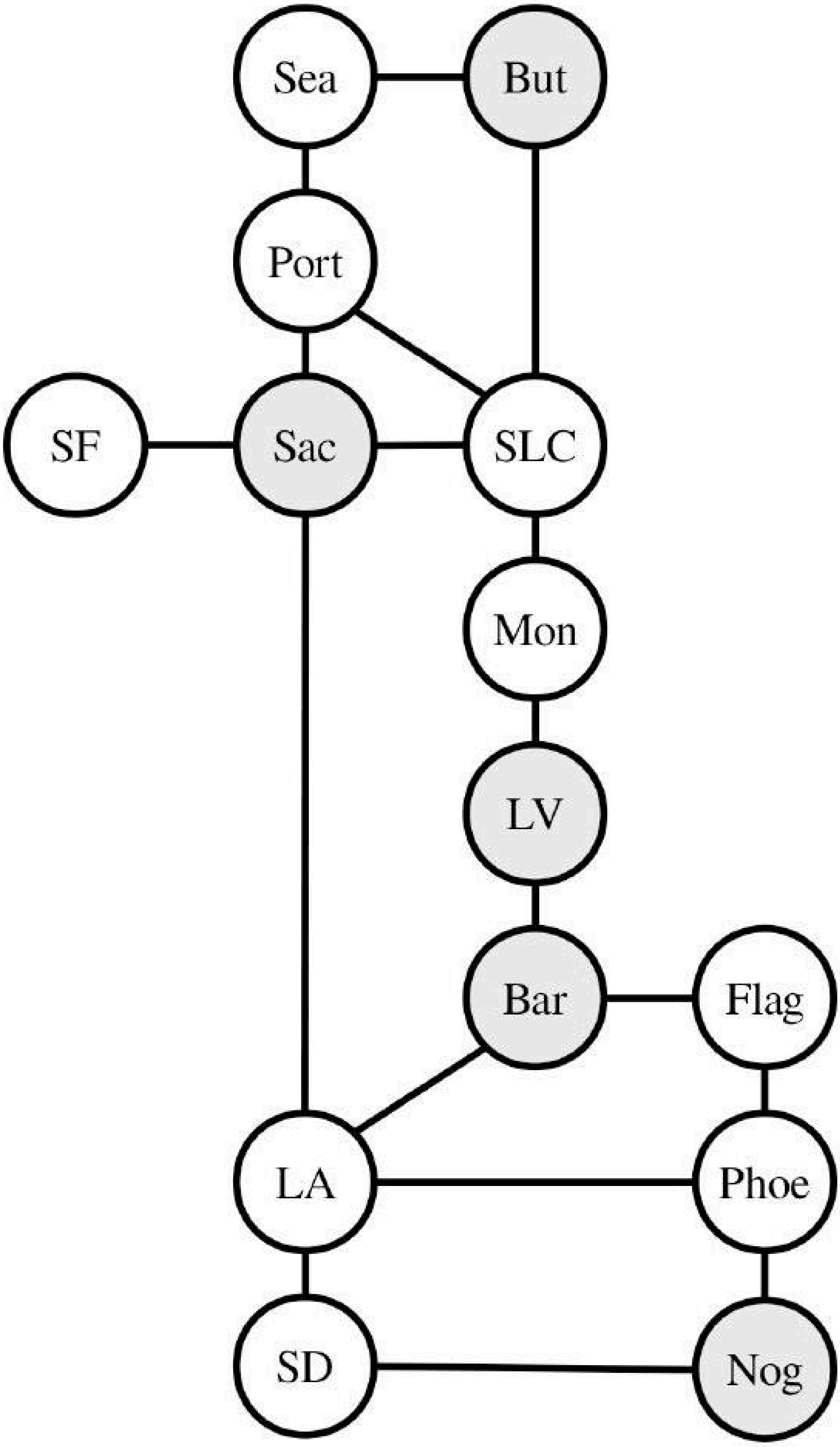
自然灾害和非自然灾害都不可避免，城市需要做好应对准备。

One problem: stockpiling emergency resources can be really, really expensive. As a result, it's reasonable to have only a few cities stockpile emergency resources, with the plan that they'd send those resources from wherever they're stockpiled to where they're needed when an emergency happens. The challenge with doing this is to figure out where to put resources so that (1) we don't spend too much money stockpiling more than we need, and (2) we don't leave any cities too far away from emergency supplies.

一个问题是：储备应急资源真的非常非常昂贵。因此，合理的做法是只让少数城市储备应急资源，并计划在紧急情况发生时将这些资源从储备的地方送到需要的地方。这样做的挑战在于如何确定资源的投放位置，以便（1）我们不会花太多钱储备超过我们需要的资源，（2）我们不会让任何城市离应急物资太远。

Imagine that you have access to a country's major highway networks. We can imagine that there are a number of different cities, some of which are right down the highway from others. To the right is a fragment of the US Interstate Highway System for the Western US. Suppose we put emergency supplies in Sacramento, Butte, Las Vegas, Barstow, and Nogales (shown in gray). In that case, if there's an emergency in any city, that city either already has emergency supplies or is immediately adjacent from a city that does. For example, any emergency in Nogales would be covered, since Nogales already has emergency supplies. San Francisco is covered by Sacramento, Salt Lake City is covered by both Sacramento and Butte, and Barstow is covered both by itself and by Las Vegas.

想象一下，你可以进入一个国家的主要高速公路网。我们可以想象有许多不同的城市，其中一些城市与另一些城市就在高速公路上。右图是美国西部州际公路系统的一个片段。假设我们在萨克拉门托、布特、拉斯维加斯、巴斯托和诺加莱斯（灰色显示）放置了应急物资。在这种情况下，如果任何城市发生紧急情况，该城市要么已经拥有应急物资，要么紧邻拥有应急物资的城市。例如，在诺加利斯发生的任何紧急情况都会被包括在内，因为诺加利斯已经有了应急物资。旧金山由萨克拉门托负责，盐湖城由萨克拉门托和布特负责，巴斯托由其本身和拉斯维加斯负责。



Although it's possible to drive from Sacramento to San Diego, for the purposes of this problem the emergency supplies stockpiled in Sacramento wouldn't provide coverage to San Diego, since they aren't immediately next to one another.

虽然可以从萨克拉门托开车到圣地亚哥，但就这个问题而言，萨克拉门托的应急物资储备并不能覆盖圣地亚哥，因为它们并不紧邻。

We'll say that a country or region is disaster-ready if it has this property: that is, every city either already has emergency supplies or is immediately down the highway from a city that has them. Your task is to write a function

如果一个国家或地区具有这样的属性，我们就可以说这个国家或地区已经为灾难做好了准备：也就是说，每个城市要么已经拥有应急物资，要么就在拥有应急物资的城市的公路边上。您的任务是编写一个函数

```
bool canBeMadeDisasterReady(const Map<string, Set<string>>& roadNetwork,
     int numCities, Set<string>& locations);
```

that takes as input a Map representing the road network for a region (described below) and a number of cities that can be made to hold supplies, then returns whether it's possible to make the region disaster-ready by placing supplies in at most numCities cities. If so, the function should then populate the argument locations with all of the cities where supplies should be stored.

该函数的输入是一张代表某地区道路网络的地图（如下所述）和若干个可以存放物资的城市，然后返回是否可以通过在最多数量的城市中放置物资来使该地区为灾难做好准备。如果可以，函数就会在参数 locations 中填入所有应存放物资的城市。

In this problem, the road network is represented as a map where each key is a city and each value is a set of cities that are immediately down the highway from them. For example, here's a small fragment of the map you'd get from the above transportation network:

在这个问题中，公路网被表示为一张地图，其中每个键都是一个城市，每个值都是一组紧邻公路的城市。例如，以下是上述交通网络地图的一小部分：

```
"Sacramento": {"San Francisco", "Portland", "Salt Lake City", "Los Angeles
"San Francisco": {"Sacramento"}
"Portland": {"Seattle", "Sacramento", "Salt Lake City"}
```

As in the first part of this assignment, you can assume that locations is empty when this function is first called, and you can change it however you'd like if the function returns false.

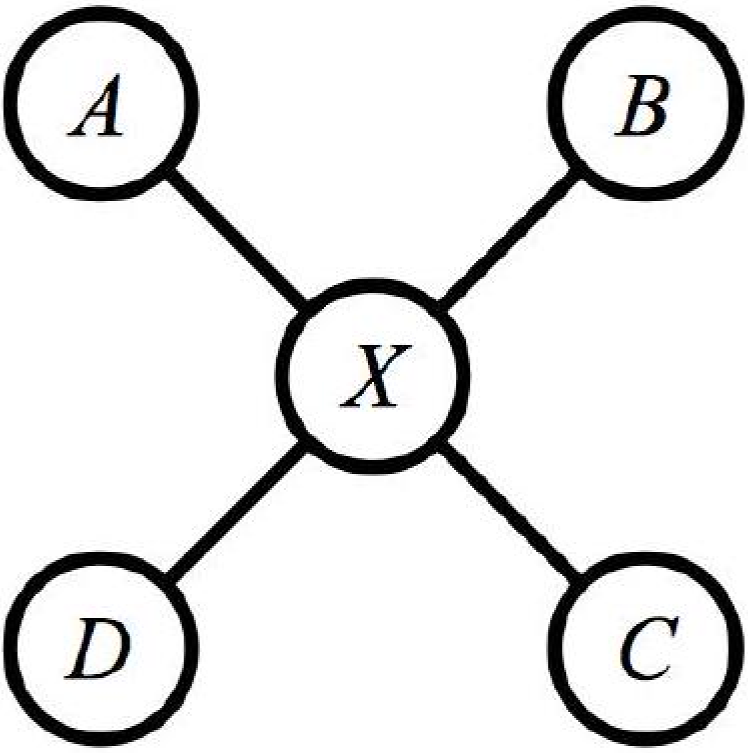
与作业的第一部分一样，在第一次调用此函数时，可以假设 locations 为空，如果函数返回 false，则可以随意更改。

There are many different strategies you can use to solve this problem, and some are more efficient than others. For example, one option would be to treat this problem as a subsets problem, trying out each subset of cities of the given size and seeing whether any of them would make all cities disaster ready. This option works, but it will be extremely slow on some of the larger test cases where there are over thirty cities - so slow in fact that your program might never give back an answer. That's not good enough.

有许多不同的策略可以用来解决这个问题，有些策略比其他策略更有效。例如，一种方法是将这个问题视为子集问题，尝试给定规模的每个城市子集，看看其中是否有任何一个子集能让所有城市都做好灾难准备。这种方法可行，但在一些有 30 多个城市的大型测试案例中，速度会非常慢，慢到你的程序可能永远不会给出答案。这还不够好。

Here's a better heuristic: Imagine there's some city X in the transportation grid that's adjacent to four neighboring cities, as shown to the right. Any collection of cities that makes this grid disaster ready is going to have to provide some kind of coverage to city X . Even if you have no idea which cities get chosen in the long run, you can say for certain that you'll need to include at least one of A, B, C, D, or X.

这里有一个更好的启发式方法：假设交通网中有某个城市 X 与四个相邻城市相邻，如右图所示。任何城市的集合都必须为 X 城市提供某种覆盖，才能使这个网格做好防灾准备。即使您不知道从长远来看哪些城市会被选中，但可以肯定的是，您至少需要包括 A, B, C, D 或 X 中的一个。



Consider approaching the problem this way: Find a city that's uncovered, then think of all the different ways you could cover it (either by choosing an adjacent city or by choosing the city itself). If you can cover all the remaining cities after making any of those choices, congrats! You're done. On the other hand, if no matter which of these choices you commit to you find that there's no way to cover all the cities, you know that no solution to your particular sub-problem exists.

可以考虑这样解决问题：找到一个没有覆盖的城市，然后想一想你可以用哪些不同的方法来覆盖它（可以选择邻近的城市，也可以选择城市本身）。如果在做出这些选择后，你能覆盖所有剩余的城市，那么恭喜你！你就大功告成了。另一方面，如果无论你做出哪种选择，你都发现没有办法覆盖所有城市，那么你就知道你的特定子问题没有解决方案。

Some notes on this problem:

关于这个问题的一些说明：

The road network is bidirectional. If there's a road from city A to city B, then there will always be a road back from city B to city A in the network, and both roads will be present in the parameter roadNetwork. You can rely on this.

道路网络是双向的。如果有一条道路从城市 A 通往城市 B ，那么网络中总会有一条道路从城市 B 返回城市 A ，并且这两条道路都会出现在参数 roadNetwork 中。您可以信赖这一点。

Every city appears as a key in the map, although some cities might be isolated (think Honolulu!) If that happens, the city will be represented by a key in the map associated with an empty set of adjacent cities.

每个城市在地图中都会出现一个键，但有些城市可能是孤立的（比如檀香山！），如果出现这种情况，该城市将由地图中与相邻城市的空集相关联的一个键来表示。

If you're allowed to use up to, say, k cities, but you find a way to solve the problem using fewer than k cities, that's fine! The set of cities you return can contain any

如果您最多可以使用 k 个城市，但您找到了使用少于 k 个城市来解决问题的方法，那也没关系！您返回的城市集可以包含任何

number of cities provided that you don't exceed k and you properly cover everything.

只要不超过 k ，并适当涵盖所有内容，就可以获得城市的数量。

The test cases we've bundled with the starter code here include some simplified versions of real transportation networks from around the world. Play around with them and let us know if you find anything interesting. Our starter code also contains an option to use your code to find the minimum number of cities needed to provide disaster protection in a region, which you might find interesting to try out once you've gotten your code working.

我们在这里将测试用例与启动代码捆绑在一起，其中包括世界各地真实交通网络的一些简化版本。如果您发现任何有趣的问题，请与我们联系。我们的启动代码还包含一个选项，可以用你的代码找出在一个地区提供灾难保护所需的最少城市数量，一旦你的代码正常工作，你可能会发现尝试一下会很有趣。

Note that some of the test files that we've included have a lot of cities in them. The provided test cases whose names start with VeryHard are, unsurprisingly, very hard tests that may require some time to solve. It's okay if your program takes a long time (say, up to two minutes) to answer queries for those maps, though if you use the strategy outlined above you should probably be able to get solutions back for them in only a matter of seconds.

请注意，我们所包含的一些测试文件中有很多城市。所提供的以 VeryHard 开头的测试用例都是非常难的测试，可能需要一些时间才能解决。如果你的程序需要很长时间（比如两分钟）来回答这些地图的查询，也没有关系，不过如果你使用上述策略，应该只需几秒钟就能得到答案。

Problem 3: Winning the Presidency

问题 3：赢得总统职位

The President of the United States is not elected by a popular vote, but by a majority vote in the Electoral College. Each state, plus DC, gets some number of electors in the Electoral College, and the person they collectively vote in becomes President. For the purposes of this problem, we're going to make some assumptions:

美国总统不是由普选产生，而是由选举团的多数票选出。每个州加上华盛顿特区在选举团中获得一定数量的选举人，他们集体投票选出的人成为总统。为了解决这个问题，我们要做一些假设：

You need to win a majority of the votes in a state to earn its electors, and you get all the state's electors if you win the majority. For example, in a small state with only 99 people, you'd need 50 votes to win all its electors. These assumptions aren't entirely accurate, both because in most states a plurality suffices and some states split their electoral votes in other ways.

您需要赢得一个州的多数选票才能获得该州的选举人，如果您赢得多数选票，您就能获得该州的所有选举人。例如，在一个只有 99 人的小州，你需要赢得 50 票才能赢得该州的所有选举人。这些假设并不完全准确，因为在大多数州，多数票就足够了，而且有些州以其他方式分配选举人票。

You need to win a majority of the electoral votes to become president. In the 2008 election, you'd need 270 votes because there were 538 electors. In the 1804 election, you'd need 89 votes because there were only 176 electors. (You can technically win the presidency without winning the Electoral College, but we'll ignore this.)

您需要赢得多数选举人票才能成为总统。在2008年选举中，您需要270张选票，因为有538名选举人。在1804年的选举中，您需要获得89张选票，因为只有176名选举人。(从技术上讲，您可以在没有赢得选举人团的情况下赢得总统职位，但我们将忽略这一点）。

Electors never defect. The electors in the Electoral College are free to vote for whomever they please, but the expectation is that they'll vote for the candidate that won their home state. As a simplifying assumption, we'll just pretend electors always vote with the majority of their state.

选举人从不叛变。选举团中的选举人可以自由投票给他们喜欢的人，但他们会投票给赢得本州选举的候选人。作为一个简化的假设，我们就当选举人总是投票给本州的多数人。

This problem explores the following question: under these assumptions, what's the fewest number of popular votes you can get and still be elected President?

这个问题探讨的是以下问题：在这些假设条件下，你能获得多少张最少的民众选票而仍然当选总统？

Imagine that we have a list of information about each state, represented by this handy struct definition:

想象一下，我们有一个关于每个状态的信息列表，用这个方便的结构定义来表示：

```
struct State {
    string name; // the name of the state
    int electoralVotes; // how many electors it has
    int popularVotes; // the number of people in that state who voted
};
```

This record contains the name of the state, its number of electors, and its voting population. Your task is to write a function

该记录包含州名、选举人数量和投票人口。您的任务是编写一个函数

```
MinInfo minPopularVoteToWin(const Vector<State>& states);
```

that takes as input a list of all the states that participated in the election (plus DC, if appropriate), then returns some information about the minimum number of popular votes you'd need in order to win the election (namely, how many votes you'd need, and which states you'd carry in the process). The MinInfo structure is really just a pair of a minimum popular vote total plus the list of states you'd carry in the course of reaching that total:

的MinInfo结构，它的输入是参加选举的所有州的列表（如果合适，加上华盛顿特区），然后返回一些关于你赢得选举所需的最低民众选票数的信息（即你需要多少选票，以及在此过程中你将携带哪些州的选票）。MinInfo 结构实际上只是一对最小普选总票数加上你在达到该总票数的过程中将携带的州的列表：

```
struct MinInfo {
    int popularVotesNeeded; // how many popular votes you'd need
    Vector<State> statesUsed; // the states you'd win in getting those votes
};
```

To implement this function, we strongly recommend implementing a helper function that answers the following question:

为实现这一功能，我们强烈建议使用一个辅助函数来回答以下问题：

What is the minimum number of popular votes needed to get at least V electoral votes, using only states from index i and above in the Vector?

如果只使用向量中指数 i 及以上的州，获得至少V张选举人票所需的最少普选票数是多少？

Notice that if you solve this problem with V set to a majority of the total electoral votes and with $i = 0$, then you've solved the original problem (do you see why?). We strongly recommend making your original function a wrapper around a helper function that solves this specific problem.

请注意，如果您将 V 设置为选举人票总数的多数，并使用 $i = 0$ 解决了这个问题，那么您就解决了最初的问题（您明白为什么吗？）我们强烈建议将你的原始函数封装在一个能解决这个特定问题的辅助函数周围。

In the course of solving this problem, you might find yourself in the unfortunate situation where, for some specific values of V and i, there's no possible way to get V votes using only the states from index i and forward. For example, if you're short 75 electoral votes and only have a single state left, there's nothing that you can do to win the election. In that case, you may want to have this helper function return some kind of sentinel value indicating that it's not possible to secure that many votes. We recommend using the special value INT_MAX, which represents the maximum possible value that you can store in an integer. The advantage of this sentinel value is that you're already planning on finding the strategy that requires the fewest popular votes, so if your sentinel value is greater than any possible legal number of votes, always choosing the option that requires the minimum number of votes will automatically pull you away from the sentinel value.

在解决这个问题的过程中，你可能会发现自己处于这样一种不幸的境地：对于 V 和 i 的某些特定值，没有可能只用索引 i 及以后的州来获得 V 票。例如，如果你缺少 75 张选举人票，而且只剩下一个州，那么你就没有办法赢得选举。在这种情况下，你可能想让这个辅助函数返回某种哨兵值，表明不可能获得那么多选票。我们建议使用特殊值 INT_MAX，它代表了可以存储在整数中的最大可能值。使用这个前哨值的好处是，你已经在计划寻找需要最少民众投票的策略了，所以如果你的前哨值大于任何可能的法定票数，总是选择需要最少票数的选项就会自动远离前哨值。

When you first implement this function, we strongly recommend testing it out using the simplified test cases we've provided you in the test framework's main menu. These test cases use real election data, but only consider ten states out of a larger election. That should make it easier for you to check whether your solution works on smaller examples.

在您首次执行该功能时，我们强烈建议您使用我们在测试框架主菜单中提供的简化测试用例进行测试。这些测试用例使用了真实的选举数据，但只考虑了大型选举中的十个州。这将使您更容易在较小的示例中检查您的解决方案是否有效。

Without using memoization, it's almost guaranteed that your code won't be fast enough. To scale this up to work with actual elections data, you'll need to work memoization into your solution. The good news is that, if you've followed the strategy we've outlined above, you should find that it's relatively straightforward to introduce memoization into your solution. Once you've gotten that working, try running your code on full elections data. I think you'll be pleasantly surprised by how fast it runs!

如果不使用 memoization，几乎可以保证你的代码不会足够快。要将其扩展到实际的选举数据中，您需要在解决方案中加入 memoization。好消息是，如果您遵循了我们在上文概述的策略，您应该会发现在解决方案中引入备忘录化是一件相对简单的事情。一旦你成功了，请尝试在完整的选举数据上运行你的代码。我想你会对它的运行速度感到惊喜！

Here are some general notes on this problem:

下面是关于这个问题的一些一般性说明：

The historical election data - and our reduced test cases - do not always include all 50 current US states plus DC, either because those states didn't exist yet, or DC didn't have the vote, or because those states didn't participate in the election, so you shouldn't assume you'll get them as input.

历史选举数据--以及我们缩减的测试用例--并不总是包括当前美国所有 50 个州和华盛顿特区，要么是因为这些州还不存在，要么是因为华盛顿特区没有投票权，要么是因为这些州没有参加选举，所以你不应该假设你会得到这些数据作为输入。

The total number of Electoral College votes to win the election depends on the number of electors, which varies over time. Although you currently need 270 electoral votes to become President, you should not assume this in your solution.

赢得选举的选举人团选票总数取决于选举人的数量，而选举人的数量随时间而变化。虽然您目前需要 270 张选举人票才能成为总统，但您不应该在解决方案中假设这一点。

Remember that in all elections you need strictly more than half the votes to win. If there are either 100 or 101 people in a state, you need 51 votes to win its electors. If there are 538 or 539 total electoral votes, you'd need 270 electoral votes to become president.

请记住，在所有选举中，您需要严格超过半数的选票才能获胜。如果一个州有 100 或 101 人，您需要 51 票才能赢得该州的选举人票。如果选举人票总数为538或539，那么您需要270张选举人票才能成为总统。

You can represent the table in the memoization step in a number of different ways. One option that isn't mentioned in the textbook is the SparseGrid type, which depending on your approach might be nice to know about. Check the documentation up on the CS106X course website for more information.

在 memoization 步骤中，可以用多种不同的方式表示表格。教科书中没有提到的一种选择是 SparseGrid 类型，根据你的方法，了解这种类型可能会很有帮助。更多信息请查看 CS106X 课程网站上的文档。

Right before the 2016 election, NPR reported that 23% of the popular vote would be sufficient to win the election, based on the 2012 voting data. They arrived at this number by looking at states with the highest ratio of electoral votes to voting population. This was a correction to their previously reported number of 27%, which they got by looking at what it would take to win the states with the highest number of electoral votes. But the optimal strategy turns out to be neither of these and instead uses a blend of small and large states. Once you've arrived at a working solution, try running it on the data from the 2012 election. What percentage of the popular vote does your program say would be necessary to secure the presidency? [2]

就在2016年大选之前，美国国家公共电台（NPR）报道称，根据2012年的投票数据， 23% 的普选票数足以赢得大选。他们是通过考察选举人票与投票人口比例最高的州得出这一数字的。这是对他们之前报告的 27% 数字的修正，他们是通过考察赢得选举人票数最多的州所需的票数得出这个数字的。但事实证明，最佳策略并非如此，而是将小州和大州结合起来。得出可行方案后，请尝试在2012年大选的数据上运行它。你的程序认为需要多大比例的民众选票才能确保总统宝座？ [2]

[1] Note that we've changed the time the assignment is due to $11:59$pm instead of $5:00$pm. We're doing that so students in Wednesday afternoon's discussion sections aren't distracted by a looming deadline.

Also, you are strongly discouraged from taking any late days on this assignment, since doing so will impede your ability to prepare for the midterm on Thursday, October 24th at 7:00 pm. ()

[1] 请注意，我们已将作业的提交时间从 $5:00\text{pm}$ 改为 $11:59\text{pm}$ 。我们这样做是为了让参加周三下午讨论组的学生不至于被迫在眉睫的截止日期分心。另外，我们强烈建议你们不要在作业上拖拉，因为这样会影响你们准备 10 月 24th 日星期四晚上 7:00 的期中考试。

[2] The historical election data here was compiled from a number of sources. In many early elections the state legislatures decided how to choose electors, and so in some cases we extrapolated to estimate the voting population based on the overall US population at the time and the total fraction of votes cast. This may skew some of the earlier election results. However, to the best of our knowledge, data from 1868 and forward is complete and accurate. Please let us know if you find any errors in our data!

[2] 这里的历史选举数据是根据多种资料整理而成的。在许多早期选举中，州立法机构决定如何选择选举人，因此在某些情况下，我们根据当时美国的总人口和投票总数推算出投票人口。这可能会使一些早期的选举结果出现偏差。不过，据我们所知，1868 年及以后的数据是完整和准确的。如果您发现我们的数据有任何错误，请告知我们！