**Introduction**

The website <http://bricklink.com> is an online marketplace for Lego. As of July 2015, there are over 7,700 stores with over 250,000,000 items listed for sale. You can buy and sell sets or individual parts, with the latter perhaps being the most important feature. Users can create wanted lists for the items they are interested in buying. Although the website offers some tools to assist in the buying process, there is no built-in mechanism to help them determine what store(s) they should buy the items from. This presents two challenges. The first is finding a set of stores that have all of the items on your list. The second is finding such a set that results in the cheapest total cost.

This is where the Brickficiency software (<http://www.buildingoutloud.com/brickficiency.php>) comes in. It was written by someone who was interested in providing this functionality. It works pretty well and has some great features—including creating wanted lists for the sets of items it finds which makes it pretty easy to actually buy the items from the various stores in the solution. Unfortunately the software has some limitations. It only allows you to find solutions that involve no more than 5 stores (we’ll call a solution involving *k* stores a *k*-store solution). Further, it can take a very long time to find solutions depending on how many items are on your list, how many stores have those items, and several other factors.

As a real life example, in the spring of 2015 I wanted to buy about 460 different parts, with quantities of each part varying from 10 to 100. I tried to use Brickficiency but after running it overnight it did not yield any solutions. I then spent over 20 hours trying to determine the best stores to buy these items from by hand. In the end I had to purchase the items from 50 different stores! No wonder the software couldn’t find a solution.

The goal of this project is to improve the algorithm used in the Brickficiency software. We will explore several different types of algorithms that might be used and discuss the limitations these (and perhaps all) algorithms have to solve this problem. We will address questions such as the following:

1. For how large of a value of *k* can we expect to find the absolute best (i.e. cheapest) *k*-store solution in a reasonable amount of time?
2. How large of a list of items can we handle reasonably?
3. Can we develop an algorithm that will find solutions when the smallest possible *k* is large (e.g. more than 10)? If so, how close to optimal can we get?
4. Can we develop an algorithm that will find *most* of the items on our list in a reasonable amount of time? If so, how do we decide which items are “acceptable losses” (that is, which items we can do without)?
5. Will any algorithm we develop find a decent solution (or *any* solution) for the case I mentioned above that ultimately required on the order of 50 stores?
6. What data structures and algorithmic techniques can we use to come up with these algorithms?
7. Can we either find an efficient algorithm to solve the problem or prove that no such algorithm exists?

**Experimenting with Brickficiency**

1. Go to <http://bricklink.com> and create an account.
2. Go to <http://www.buildingoutloud.com/brickficiency.php> and download the latest version of the software (Under the **Download** heading, not under the Previous Versions heading).
3. Open the zipfile and extract the executable file from it, placing it on your desktop or somewhere else where you can find it.
4. Download the following file and extract it on your desktop (or somewhere you can find it): <http://www.cs.hope.edu/~cusack/ShowFiles/Assignments/Bricklink/WantedLists.zip>.
5. Run the executable. It will take several seconds to load. Wait for the message to say “Loading…Done”.
6. Go to **File🡪Open** and select ***NarrowPlates.bsx*** file from the **WantedLists** directory you just extracted from the zipfile. Your screen should now look something like the one to the right🡪
7. Go to the **Tools** menu and click the **Calculate** button. It should pop up the window shown here🡪
8. Set **Maximum combinations to query** to 3. Then check the box that says **Log in to retrieve Price Guide pages** and enter your bricklink username in the **Username** box. Finally, click **Calculate**.
9. Enter your bricklink password when asked.
10. The algorithm will now try to find the cheapest set of up to 3 stores that all of the items from the wanted list can be obtained from. After about 4 minutes it should pop up a web-browser window that contains the results. If you have the settings as above, it should give 10 solutions—5 with two stores and 5 with 3 stores. Notice that for each solution it gives the total cost, the cost from each stores, and a list of what items to buy from which stores, including the cost of each item. Also notice that the bit +/- icons can be used to show/hide the details of the various solutions.

 **Obtaining the Source Code**

1. Create a new folder in the *My Documents* folder using your VanderPlex user name (e.g. *ryan.mcfall*). This is where the local copy of your files will be stored.
2. Start **Visual Studio**.
3. Use **View->Repository Explorer** to open the Repository Explorer window. [[1]](#footnote-1)
4. Navigate to *https://svn.hope.edu/CSCI/cusack*.
5. Select the folder *Brickficiency* and click the **Copy to** icon in the repository explorer toolbar:
 
6. Navigate your SVN repository (*https://svn.hope.edu/CSCI/students/YOUR.NAME*) and select your folder and click OK twice.
7. Now the solution is located in your student repository. Next we need to download it onto your computer (you can repeat this on other computers later if you wish).
	1. From the *File* menu, select **Subversion🡪Open from Subversion**.
	2. For the URL, enter **https://svn.hope.edu/CSCI/students/YOUR.NAME**.
	3. Double-click **Brickficiency** and then **Brickficiency.sln.**
	4. Change **Local Directory** to the directory within *My Documents* you created above and click **OK**.
	5. It should download the project to your computer and in a minute or so you should be able to explore the files.
8. In the *Solution Explorer* tab, click the little triangle next to **Main.cs** and then double-click **MainWindow**.
9. Make sure **classroomUseMode** is set to **true** (on/near line 119) and save the file.
10. When you run it you will notice that there are a few differences from the “official” version. Most notably, the Calculate menu item now has a few submenus. Hopefully you can figure out what each one does. If not, ask.
11. Open up the **NarrowPlates.bsx** file and run the **Best Solution**. You will notice that this now runs the 3-store solution in around 8 seconds (not including the time it takes to download the prices). This runs my algorithm to solve the problem.
12. You will notice there is also a **Best Solution (Slower)** algorithm. It is slightly slower than the **Best Solution**, but it is based on the original algorithm so I have left it so you can see two different (but many ways similar) approaches to solving the problem.
13. The **Approximate Solution** will run the exact same algorithm as **Best Solution** except that it limits how long it lets it run for each value of *k*. Run it for values of *k* from 2 to 8 with a 30 second time limit and take a look at the results. Some of them might surprise you.
14. The **Partial Solution** will run an algorithm very similar to Best Solution except that it allows solutions that have only *some* of the items on the list and it allows you to limit how long it will run for each value of *k*. Try running it on **BiggerList.bsx** with *k* values from 4 to 8.
15. The three menu items that end in (Custom) are where you come in. None of them are implemented yet. In the near future you will implement those algorithms.
16. Finally, you will want to upload your changes to SVN (you have only made one, but we need to practice this since you will need to get in the habit of doing it regularly). To do so, in the *Solution Explorer* tab, right-click **Brickficiency** (the line right under Solution) and select **Source Control🡪Commit Project Changes**. You can enter a comment to help you track your changes (e.g. “I changed the classroomUseMode variable to true”) and then click OK. (I think you can do this by also right-clicking on the Solution and selecting **Commit Solution Changes**.)
17. If you work on multiple machines, you will want to make sure you always commit your changes when you are done working. But you also need to update each machine with the latest version before you start working. To do this, right-click on the project (Brickficiency) and select **Source Control🡪Update Project to Latest Version**.
18. Always get in the habit of updating as soon as you start working and committing as soon as you are done. That way you won’t make changes to old code and then have problems when SVN tries to merge your code.

**A few things to know about C#**

Much of the syntax of C# is similar to Java, but there are some differences. As you would expect, the available classes are different, although there are similarities, too. The most relevant things you should know are outlined next (at least the ones I thought of as I was writing this).

* The **List** class is equivalent to Java’s ArrayList, although it uses the syntax for array indexing instead of a get method. To add an item to a List, use the List.Add() method. To retrieve the ith element of the list, use brackets like you would in an array (e.g. myList[i]).
* The **Dictionary** class is similar to Java’s Hashmap. Use the Dictionary.Add() method to add a new item to the Dictionary. To retrieve a value from a Dictionary, treat it like an array and use brackets (e.g. myDictionary[key]) regardless of what the argument type is.
* You can split a class up among several files. Many of the files you will look at, including **Main.cs**, **Calculations.cs** and **Algorithms.cs**, are all part of the same class. In this case it would have been better if these were all separate classes, but I didn’t write the original version so it is what it is. But this does allow you to partition a class into several files so that each file can have the most relevant methods/fields to a particular part of the class. This is why **Algorithms.cs** has been created—most of the code you will care about is there.
* You will notice some **Parallel.For** loops in the code. These are a really convenient way to parallelize your code with almost no effort. These loops run each iteration on a separate thread in an undetermined order. This is really only useful if your machine has multiple cores. Thus, they are only applicable in situations where the iterations of the loop are independent of each other. If you use these, you need to be careful about not modifying the same variable(s) in different iterations of the loop. Much more could be said about this, but the bottom line is that “with power comes responsibility.” In other words, you don’t get something for nothing.

**Relevant classes**

The classes that you need to pay the most attention to are the following.

* **Item**: This represents an item on the wanted list. **Item** has a slew of public fields. Your algorithm will probably only use a couple of fields: **extid** (the unique string identifier of the item) and **qty** (the desired quantity of the item). If you find any of the other fields to be relevant to your algorithm, feel free to use them.
* **StoreItem**: A **StoreItem** contains much less information than a regular item. It contains only two public fields: **price** and **qty**. In this case, **qty** is the quantity that the store has in stock of this particular item. Each store has a Dictionary that maps **extids** to **StoreItems**.
* **Store**: A **Store** contains two fields—the **name** of the store (a string) and a **Dictionary<string, StoreItem>** called **itemList** that stores all of the items available in the store. They keys are the **extids** of the items.
* **FinalMatch**: A **FinalMatch** is the class used to store the data regarding a list of stores which can together satisfy the requirements specified by the Wanted List. Every time you algorithm finds a set of stores that work, it should call the method **addFinalMatch(List<string> storeNames)**. This will create a **FinalMatch** object that will be tested against other solutions. The code automatically saves the best *n* solutions, where *n* is the number chosen in the *Matches per combination* field. Besides calling this method, you don’t need to know a whole lot about this class (at least not for now).

You will notice that these classes have mostly public fields. Although this is not great in terms of object oriented design, it makes things a little easier for your algorithm and it will actually make everything go a little faster, which in this case is a good thing.

**Relevant fields/parameters**

You should spend most of your time in the **Algorithms.cs** file. It contains the most relevant methods. In particular, it contains the existing algorithms and the methods you will need to implement for your algorithm. Here are some details about the parameters to those methods.

* int k The number of stores to use in the solution.
* List<Item> **itemList**: The items on the Wanted List. The syntax should make it clear that this list is storing objects of type **Item.**
* List<Store> **storeList**: A list of the available stores. Luckily for you, the list only contains stores that have at least one item from the wanted list. In other words, the non-relevant stores have already been removed for you.

**The Algorithm**

In case it isn’t clear, the goal of the algorithms is to find subsets of *k* stores from *storeList* that together have the required number of each of the items on *itemList* and have those items as cheap as possible. Note that this does not necessarily mean that any individual items will be cheaper at any of these stores than they are from other stores. In fact, it is possible that *all* of the items are available cheaper from other stores.

**Implementing your algorithms**

Before you implement any algorithms you should spend some time looking at the existing algorithms to get a feel for how they work, how they interact with the “behind the scenes” code, become more comfortable with C# and Visual Studio, etc. In particular, look at KStoreCalc, OneStoreCalc, TwoStoreCalc, etc.

**Running the algorithm on Cached Data**

One problem with working with live data is that it is always changing. In fact, it is possible that you run the algorithm twice within a few minutes and get different solutions because one or more products were purchased (so they are no longer available) or added to a store’s inventory, and/or stores have been added or removed. To make it easier to debug your code and compare your results with others, you can test your algorithm using cached data. To do so, follow these steps:

1. Go to your Project directory and go to **Brickficiency\bin\Debug**\
2. Remove any files from **PGCache**
3. Copy everything from **PGCache2** to **PGCache**
4. Move the files **bfdb.zip** and **bfdb.sdf** from **PGCache** to your roaming profile directory—probably something like **C:\Users\YOURNAME\AppData\Roaming\Brickficiency**. If you can’t find this directory, get help. Overwrite the files that are already there.
5. Restart the application. *If you are asked to update the database because it is “X days old”, say* ***NO****!* Saying “Yes” will overwrite the bfdb.zip file you just copied.
6. When you run the algorithm, make sure the option **Log in to retrieve Price guide pages** is *NOT* checked.
7. Now when you run the algorithms, it will used the cached data. This will allow you to compare your results with the results of others. You can only do this with the wanted lists that were originally given to you unless you re-generate the cached data.

If you want to generate new cached data, you can do so as follows (you might not ever do this):

1. Uncomment lines 227-229 from **Calculations.cs** (in method GetPGPages). In case the lines number have changed, find these three lines and uncomment them:
if (!Directory.Exists("PGCache")) { Directory.CreateDirectory("PGCache"); }

System.IO.StreamWriter file = new StreamWriter("PGCache\\"+item.type+"\_"+
 item.number+"\_"+item.colour+".txt");
file.Write(page);

1. You will probably want to delete the files **bfdb.zip** and **bfdb.sdf** from your roaming profile so it will download the newest ones.
2. Delete all of the files in PGCache (or copy them to another directory).
3. Restart the application. It will probably take a several seconds to download and extract the database. I sometimes have problems with this step. If it seems to do nothing for more than a few minutes, quit and try again. Eventually it should succeed. (I think the problem might be that the server that the file is coming from is sometimes slow, but I am not certain of this.)
4. Load the relevant wanted list(s) and run the algorithm once for each, making sure the **Log in to retrieve Price guide pages** checkbox is *checked*. It only needs to download the price guide for each item—you don’t have to let the algorithm finish. Make sure you load all of the wanted lists for which you want the data cached.
5. Re-comment lines 227-229 so you don’t overwrite the cached files.
6. Now you have all of the data you need—copy it so you can retrieve it later as needed:
	1. Copy the **PGCache** directory (Brickficiency\bin\Debug\PGCache from your project directory) to **PGCache3**.
	2. Copy **bfdb.zip** and **bfdb.sdf** from your roaming profile to **PGCache3** so you can copy them back if/when needed.
7. Now you can run the algorithm in the future using this data if you don’t download the newest database and you uncheck the option **Log in to retrieve Price guide pages.**

**Projects/Problems**

1. **Project:** Create an account on <http://bricklink.com>. Then download and install the **BrickStore** application (available from <http://www.brickforge.de/software/brickstore>). Finally, create several wanted lists using BrickStore and upload them to your BrickLink account and use them to search for the items. This assignment will likely involve some trial-and-error and you are more than welcome to assist each other on it. You should submit your wanted lists as .bsx files. (Later we will use this type of file with **Brickficiency.**)
2. **Problem:** Create a wanted list with several items on it and a list of several stores that each have (some of) those items at various prices such that the optimal solution involving *k* > 1 stores (use whatever value of *k* that you wish) never obtains a single item on the wanted list at the cheapest possible price.
3. **Problem:** Analyze OneStoreCalc, TwoStoreCalc, FiveStoreCalc, and KStoreCalc. That is, give the *time* and *space* complexity in terms of *n* (the number of items on the wanted list), *m* (the number of stores) and *k* (the number of stores to use in the solution). Give the worst-case analysis. Make sure you take into account how long any method calls take (e.g. AddFinalMatch). Give a brief but *complete* justification of your analysis (“use your words!”).
4. **Project:** Implement CustomAlgorithm and CustomPreProcess (if necessary). Your goal is to make your algorithm as fast as possible while still returning the optimal (cheapest) solution. Your solution should work for any positive value of *k* (although it might take a *really long time* as *k* gets larger than 4 or 5). You may use ideas from the existing algorithms, but you must contribute some of your own ideas in order to improve upon those algorithms. Alternatively, you may create an algorithm from scratch. Give a complete analysis of your algorithm (the comments/suggestions from the previous problem apply to the analysis of your algorithm).
5. **Problem:** Create two wanted lists that cause problems for your algorithm and/or the “stock” algorithms. The two lists should try to reveal two *different* deficiencies of the algorithm(s). Submit your solutions as .bsx files. Include a brief write up that describes exactly what problem(s) each wanted list causes for the existing algorithms. If possible, come up with suggestions for how the algorithms might be improved to deal with your wanted lists more efficiently.
6. **Problem:** Briefly discuss whether or not each of the following algorithmic techniques can be applied to this problem. If a technique does not apply, explain why not, being as specific as possible. If a technique does apply, discuss the strengths and weaknesses of the approach.
	1. Exhaustive Search
	2. Divide-and-Conquer
	3. Dynamic Programming
	4. Greedy
	5. Backtracking
	6. Brand-and-Bound
	7. Genetic Algorithms and/or Genetic Programming
	8. Others?
7. **Project:** Implement CustomApproximationAlgorithm and CustomApproximationPreProcess (if necessary). Your goal is to make your algorithm return as good of a solution as possible in a short amount of time. Your algorithm might always finish quickly or (probably better) keep trying to find even better solutions until it is asked to quit. As with CustomAlgorithm, this should work for an arbitrary value of *k*. It is particularly important that this works for larger values of *k*. Test it using one of the larger wanted lists for values of *k* as large as 20 or 30. You might need to set the running time to several minutes in order to find *any* solutions for values of *k* this large.
8. **Project:** Implement CustomPartialAlgorithm and CustomPartialPreProcess (if necessary). Your goal is to make your algorithm return, as quickly as possible, good (i.e. cheap) solutions that include as many of the wanted items as possible. This algorithm should be viewed as a special case of an approximation algorithm. That is, it will be asked to run for a certain period of time and then it will be asked to stop.
The existing code already does part of this, although you might need to change how it works depending on your approach. Each time AddFinalMatch is called, it adds the new solution, sorts the list of solutions, and keeps the top *L* solutions, where *L* is chosen by the user in the GUI. Currently the sorted order is based first on the number of items in the solution (descending) and second by price (ascending). For instance, assume the wanted list contains 1000 items (total count of items, not different types of items) and the top five (*L*=5) items on the list have 900, 910, 920, 935, 935 items (where the top solution is the one with 935 items that has a cheaper total cost). If a solution with less than 900 items is added, it will be discarded immediately. If it has more than 900, it will cause the one with 900 items to be removed. If it has exactly 900 items, it will remain on the list only if the total cost is less than the current one that has 900 items.
If you feel the need to change how this ordering works, you will need to modify the method CompareTo from the FinalMatch class. You may need to change more than this. *In any case, you should clear any changes to the code outside of your algorithms with me before doing it!*
9. **Problem:** Do *one* of the following (your choice):
	1. Make your CustomAlgorithm run in polynomial time.
	2. Prove that no polynomial-time algorithm exists to solve the problem.
	3. Prove that the problem is NP-complete.

Assume the wanted list has *n* items, that there are *m* stores, and we want a solution involving *k* stores. Then by *polynomial-time* algorithm, we mean one that is polynomial in *n*, *m*, and *k*.

1. If *Repository Explorer* does not appear, try the following: Download and install *AnkhSVN* (version 2.5.12471 works, so get that one or the latest one available) from <https://ankhsvn.open.collab.net/downloads>. Then go to *Tools—Options—Source Control*, choose *AnkhSVN* and click OK. [↑](#footnote-ref-1)