Pre-Class Work 12: Recursive Backtracking

Recursive Backtracking [Background Reading]

□ Motivation

Let's say that we're a UW student living in the dorms. This means we have lots of dining money in our dining cards. Almost too much money... It's the end of the quarter, and we need to make sure we spend all of the money before it gets donated to HFS (who probably does not need that money). We're at the Hub and we have lots of menu items to choose from.

What are the possible choices that we can make to spend all our money, and how can we find them?

Well, we could just find out all possible combinations of items that we could have, but in some cases, that doesn't really work out, right? We have finite amounts of money, meaning finite numbers of choices, meaning that not every combination is viable. Additionally, the menu items at the Hub are finite, meaning that they can run out of stock.

So we're limited by both our money and the possible items we can buy.

Last time, we introduced **exhaustive search**, in order to look for all possible solutions to a certain problem. While this may seems useful in some cases, in most cases, we're looking for a particular solution that fits some set of conditions.

This means that we don't always need to look through every single possible choice, especially when we know that there's no point in continuing to look when that path has already broken the conditions for the solution that we want.

□ Recursive Backtracking

Instead, we should use **recursive backtracking** to go back to right before those conditions were broken and continue searching. Let's break down what **recursive backtracking** means. Here, **backtracking** means *"finding solution(s) by trying partial solutions and then abandoning them if they are not suitable"*, while **recursive** means that we're using recursion to help us do so. Remember the **choose-explore-unchoose** pattern that we introduced in **exhaustive search**? Let's revisit that in more detail.

Recursive backtracking often comes in handy for problems where we're:

- producing all permutations of a set of values

- parsing languages
- making games: anagrams, crosswords, word jumbles, 8 queens
- and in combinatorics and logic programming

You may notice some similarities of **exhaustive search** and **recursive backtracking**. And you're right, they're actually one in the same (if you use recursion for both)!

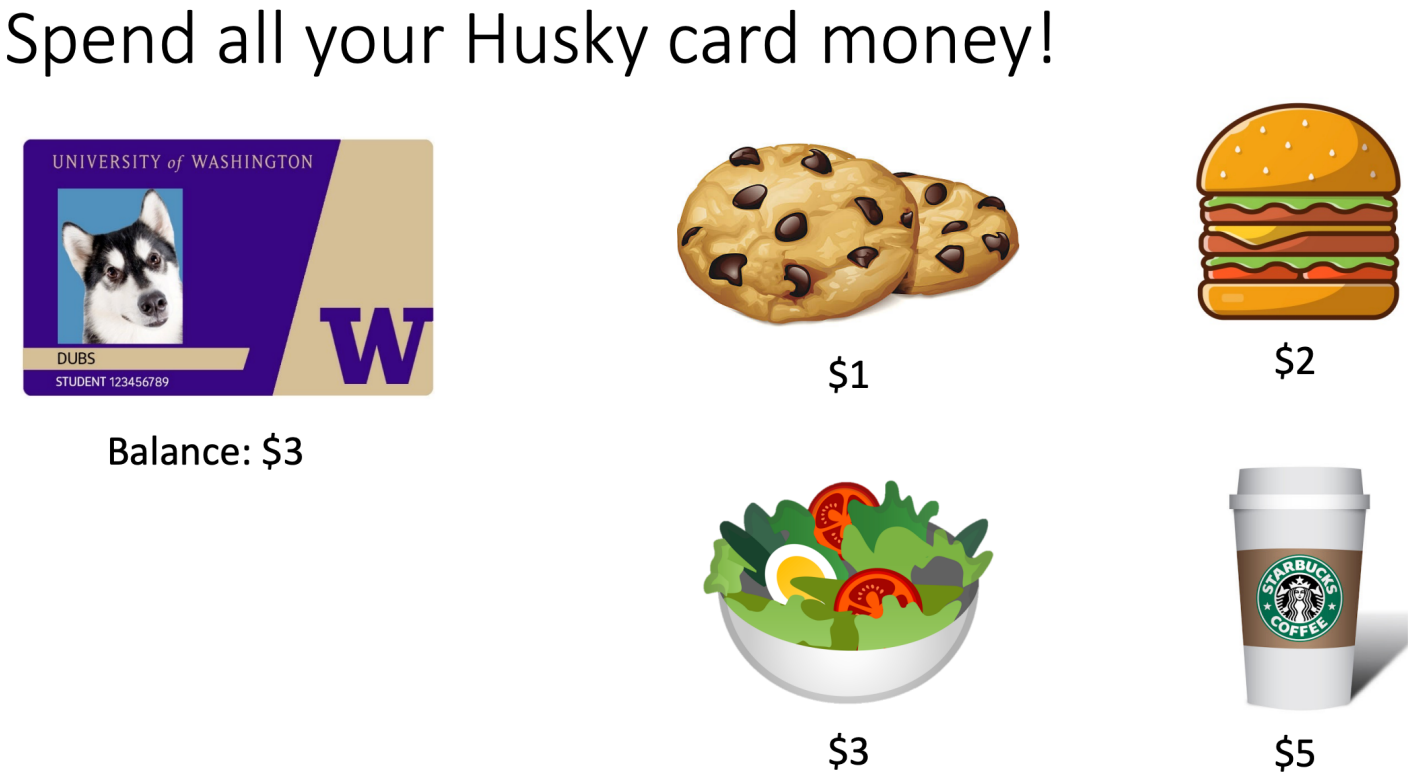
Think of **exhaustive search** as being limited by a condition of the number of choices you can make. You can't make infinite choices! But once you get to a certain number (i.e. the number of digits), you're out of choices and have to backtrack.

□ Main Points

- **Exhaustive search** aka finding all possible solutions isn't always practical when we want specific solutions under certain conditions - we can't make infinite choices!
- Instead, we can try **recursive backtracking**, which involves using **recursion** to make choices, explore their consequences, and backtrack if they lead to invalid solutions.
- We will work with the **choose-explore-unchoose** pattern when exploring **recursive backtracking**!
- **Recursive backtracking** is especially helpful when we're producing all permutations of a set of values, parsing languages, and more!
- Main idea: once we get to a certain number, we are out of choices and must backtrack.

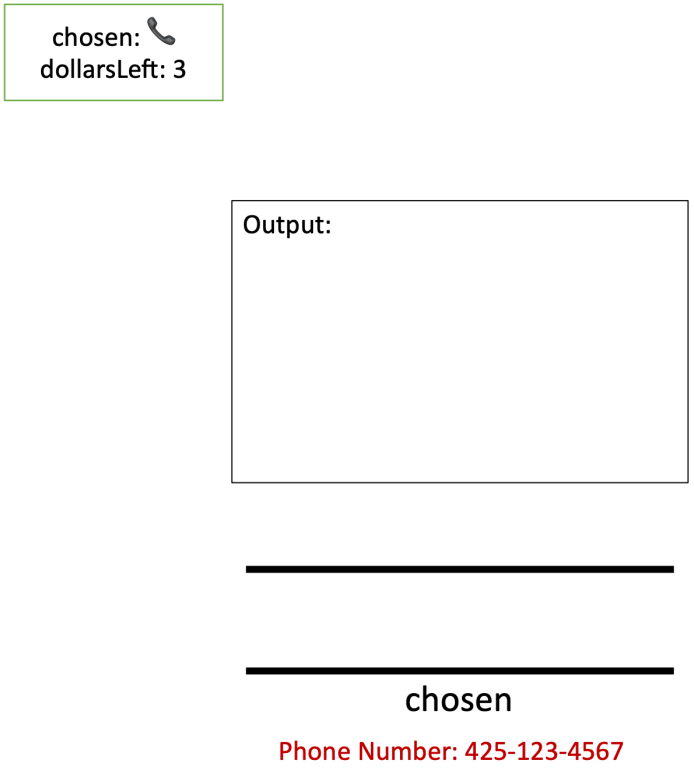
Choose-Explore-Unchoose!

Let's go back to our example with dining money and items at the Hub.

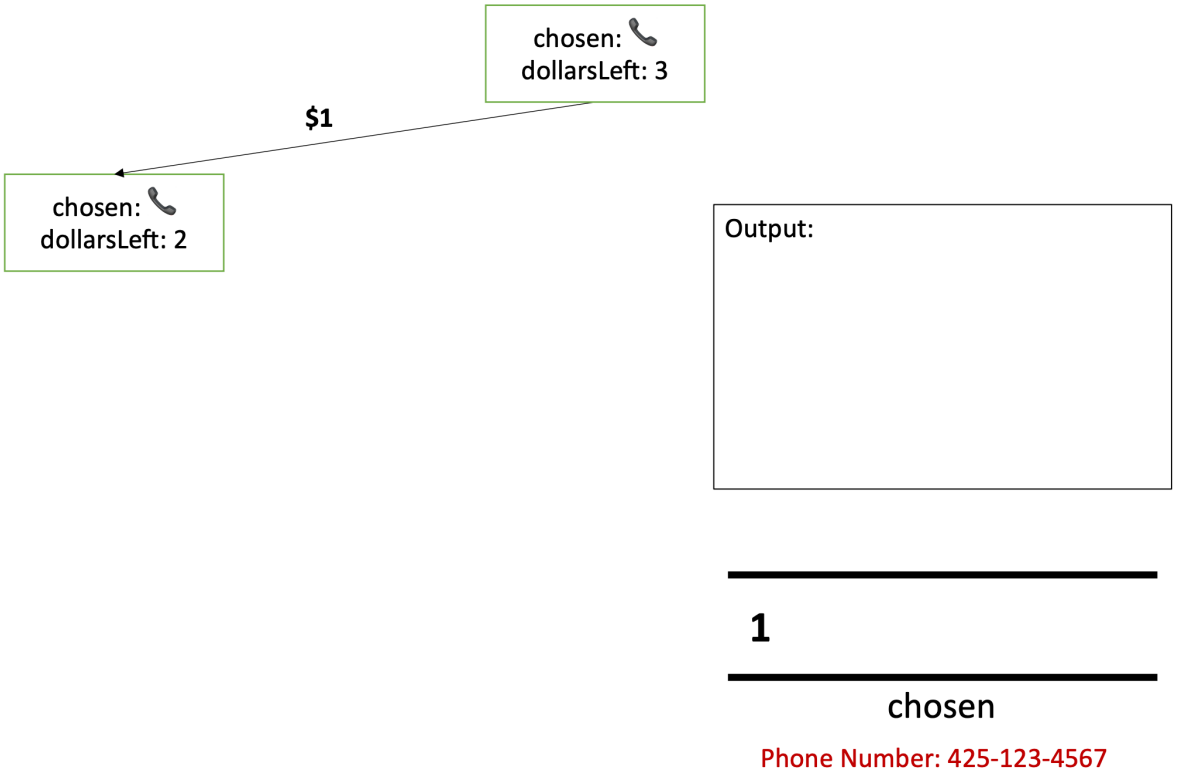


Given that our balance or `dollarsLeft` is $3, and we have the above assortment of menu items, what are all the possible combinations of food items that we can buy? Let's use **recursive backtracking** to find out!

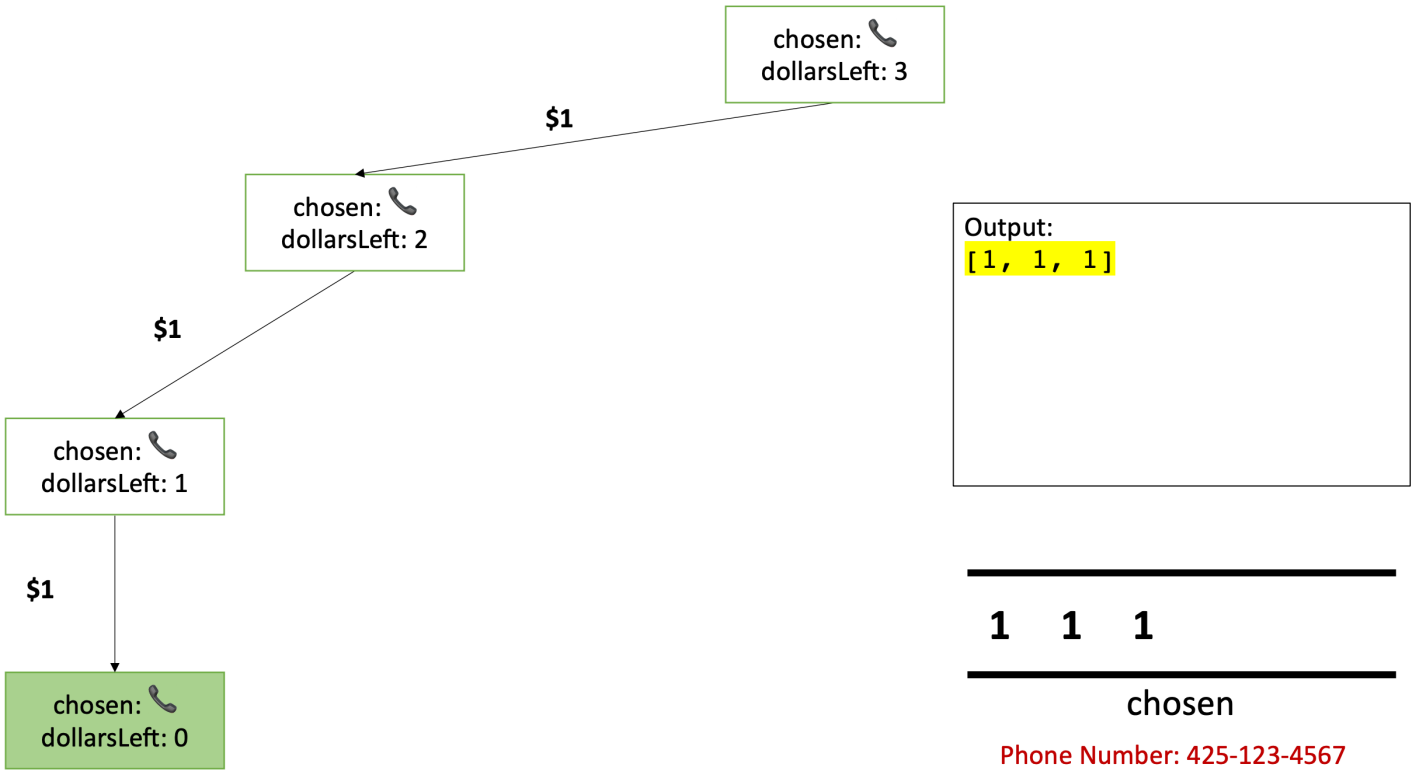
As we saw in the **exhaustive search** lesson, we need a way to remember the choices we've already made. Thus, let's create a data structure `chosen` to store them! It can be any data structure we feel best suits the situation, which in this case, could be a `List`. Here's what we're starting with:



Now let's build our decision tree. Say we choose the $1 cookie to start with. What would `dollarsLeft` and `chosen` look like?



We **choose** 1, then update `chosen` and `dollarsLeft` accordingly for that choice. Since we haven't met our base case of using all our money in our Husky Card (`dollarsLeft = 0`), we don't output anything and continue **exploring**. Let's say we choose another cookie that's $1. Then again, until our `dollarsLeft = 0`. What happens when we meet our base case?



In this event, chosen: [1, 1, 1]; output: [1, 1, 1]; dollarsLeft: 0.

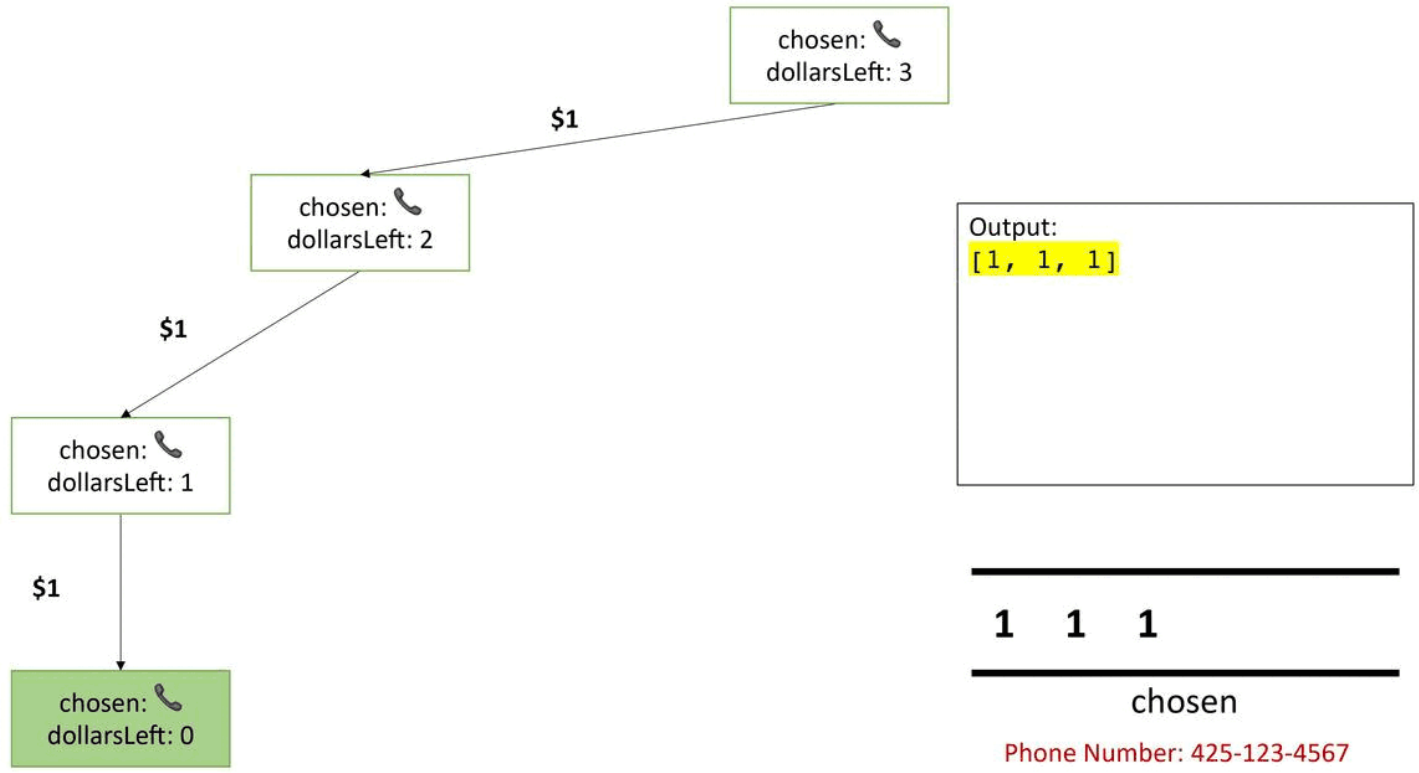
At that point, we know we've arrived at some outcome: we've used all of our dining money! Thus, we can print out what we've chosen in `chosen`. [Note that throughout all of this, `chosen` was being updated accordingly due to the fact that it's a *reference*. This means, no matter which stage we're at, which recursive call we're on, we can still access the same `chosen`.]

Great! We've reached an outcome. But we have so many other options to explore, how will we manage that if we're at a dead end? Let's hit CTRL+Z and **unchoose** the last cookie we chose to buy.

Recall that with recursive calls, once we reach a base case, we return to the instance that called the method again. This means that we go back to when `dollarLeft = 1`. In that method, we added to `chosen` as a part of our **choose** (`chosen = [1, 1, 1]`). Then after **exploring**, we return and know that at that point whatever was most recently added either worked out or it didn't.

We, in this present recursive call, don't have to worry about dealing with that; we need to focus on **unchoosing** so we're able to **choose** again!

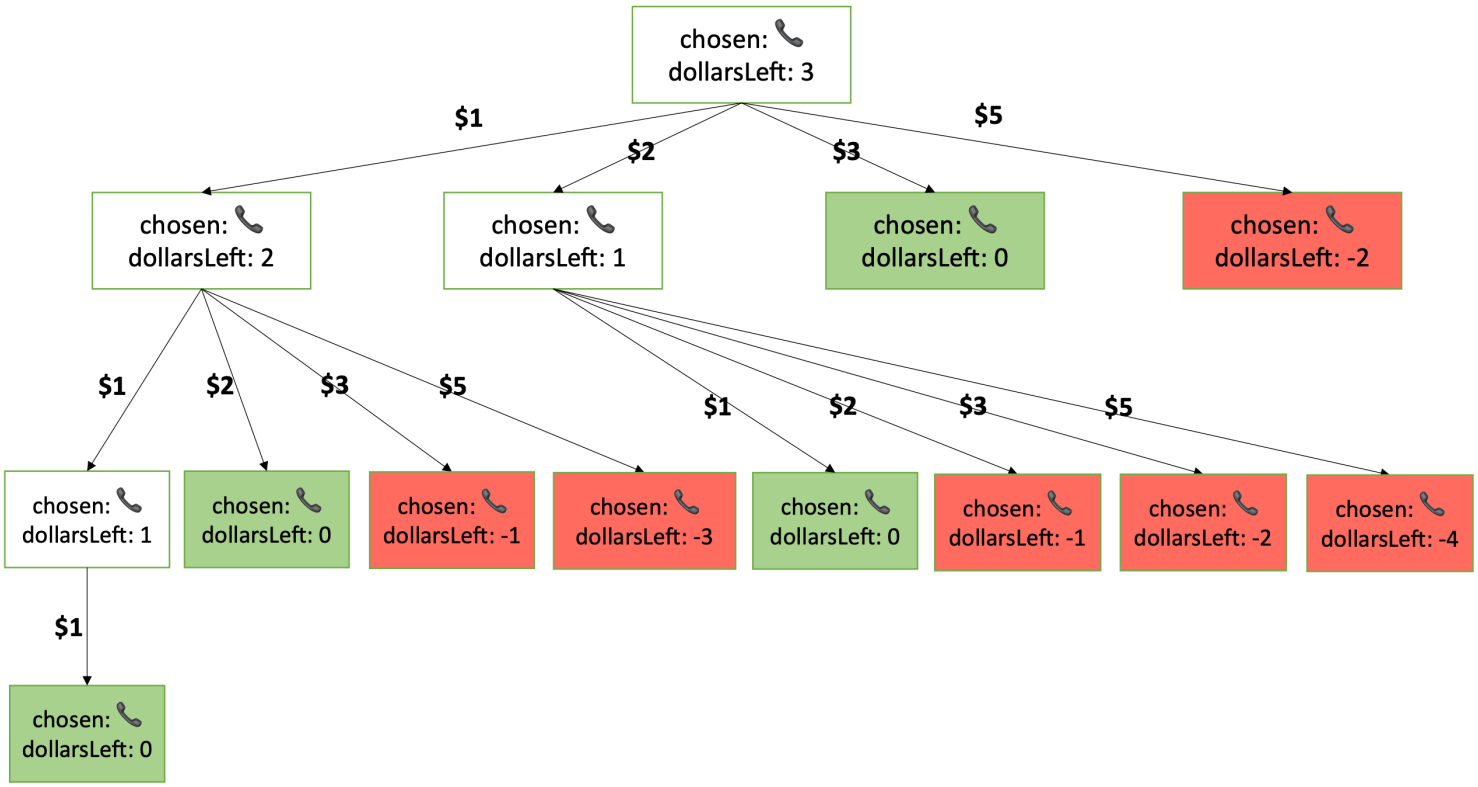
Let's instead choose to buy a $2 burger from DubStreet.



Oof. Before: {chosen: [1, 1], output: [1, 1, 1]; dollarsLeft: 1}. After: {chosen: [1, 1, 2]; output: [1, 1, 1]; dollarsLeft: -1}.

Backtrack! It broke the conditions!

1 + 1 + 2 = 4, and 3 - 4 = -1. Thus, this combination isn't viable, so we shouldn't print anything to output, and instead return to the recursive all that called it, then **unchoose** it.



The tree stops after a path meets the base case. There are 4 viable outcomes, and 6 nonviable ones.

As seen above, we repeat this pattern for all choices, to create this decision tree! All the choices in red indicate an outcome where we're in debt and thus stopped exploring. All the choices in green indicate an outcome where we used all our dining money, outputted the combination that got us there, and stopped exploring.

And finally, all the choices in white were places where are able to make more choices until we are either in the green or the red.

Main Points

- The **choose-explore-unchoose** pattern is instrumental in implementing **recursive backtracking**.
- We will repeat this pattern for all choices to make a decision tree. We will first **choose**, then **explore**, then **backtrack** aka **unchoose** once we have gone too far and have an invalid solution.
- As it is a pattern, you will find yourself writing very similar scaffolds within your code.
- However, the hardest part about **recursive backtracking** and recursion in general is knowing when and how to stop, as well as what each call is doing!

arrangements

Write a method named `arrangements` that accepts a `List of names` as a parameter and prints out **all of the possible arrangements** of the people in a line. For example, suppose a List called list stored the following names: `["Oscar", "Sumant", "Jun"]` Then a call of `arrangements(list)` should produce the following output:

```
[Oscar, Sumant, Jun]
[Oscar, Jun, Sumant]
[Sumant, Oscar, Jun]
[Sumant, Jun, Oscar]
[Jun, Oscar, Sumant]
[Jun, Sumant, Oscar]
```

The order in which you show each of the arrangements does not matter. The key thing is that your method should produce the correct overall set of arrangements as its output. You may assume that the list passed to your method is not null and that the list contains no duplicates.