Note: in these documents you will often find ⇒ next to some text that suggests doing some work. When I feel like it, I will put my answers to those suggestions in the corresponding videos. So, at times these written documents will be quite terse, because it is a huge amount of work to write up examples, as opposed to working them out with pencil and paper, or writing some code.

Introduction to the Course and Some Terminology

This course is the highest level required technical course. As such, its topic is the study of algorithms. The title is a little stupid ("Algorithms and Algorithm Analyis")—it probably should be *Advanced Algorithms*, recognizing that you've been studying algorithms in every computer science course (and other places, like work, or independent study) you've ever taken. And, of course, "analyzing" something is just part of studying that something in general. My theory is that they didn't want to scare people off (similarly to how perhaps the most challenging and/or irritating course in the major is innocently named "introduction to the theory of computation").

The idea of an *algorithm* is a fundamental concept that can't really be precisely defined (like "point" and "line" in geometry). From your previous CS courses (and life), you probably have a good sense of what it means.

> Levitin says that "an algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time." This, of course, is just defining a fundamental undefinable concept in terms of other undefinable concepts. But, we get what he means, I guess.

Some Informal Definitions

A *problem* is a collection of *instances* for which we hope to create an *algorithm* that will produce desired results when given an instance of the problem.

For example (because, just like the term "algorithm," the terms "problem" and "instance" also can't be formally defined, really), the sorting problem is "given a finite sequence of items, each with a key, where the keys can be compared, rearrange the items so the keys are in some specified order (increasing or decreasing). An instance of this problem is "sort the numbers 3, 1, 5, 7, 11, 2, 4, 101, 26 into increasing order."

To *solve* the sorting problem means to create an algorithm that can produce the correct results—a sorted list—for any given input—an arbitrary list.

Note that there are a bunch of fundamentally different algorithms that solve the sorting problem. In CS 2050 you should have already been exposed to *efficiency analysis*—that, for example, many sorting algorithms take $O(n^2)$ time to execute in the worst case (where n is the number of items being sorted), while others only take $O(n \log n)$. In this course we will do efficiency analysis more carefully and in more depth.

This course will help you to be better at

> learning a new algorithm someone else has designed,
>
> designing new algorithms (something that you do, technically, every time you create a new program) using a number of design techniques,
>
> analyzing the efficiency of a given algorithm,
>
> implementing a given algorithm, and
>
> dealing with the computational intractability of certain problems.

The Brute Force Algorithm Design Technique

We begin our advanced study of algorithms with the "brute force algorithm design technique."

> The course will mostly be designed around successive algorithm design techniques: brute force, divide and conquer, dynamic programming, greedy, transform and conquer, and branch and bound.

Brute force is actually in some sense the most generally applicable design technique, and is a good thing to start with when you are faced with a new problem. But, of course it is often less efficient than more clever algorithms. Also, brute force, as we will try to be convinced of in the upcoming discussion, is quite easy to implement.

If you are faced with a new problem and you can't see how to solve it by using brute force, then you are very unlikely to be able to come up with a cleverer ("more clever?" I just spent a few minutes looking online to see which is correct, and opinion is mixed) approach. And, in some real-world situations, brute force may be adequate. In fact, brute force is always good enough if the size of the instances of interest is small enough. Part of the benefit of our work on analysis in this course will be to realize that sometimes working a long time to come up with a more efficient algorithm just isn't worth it. And, brute force is often fairly easy to implement.

But there are sometimes more clever (just to be fair) algorithms that are more efficient. Plus, the more clever algorithms are sometimes actually easier to implement.

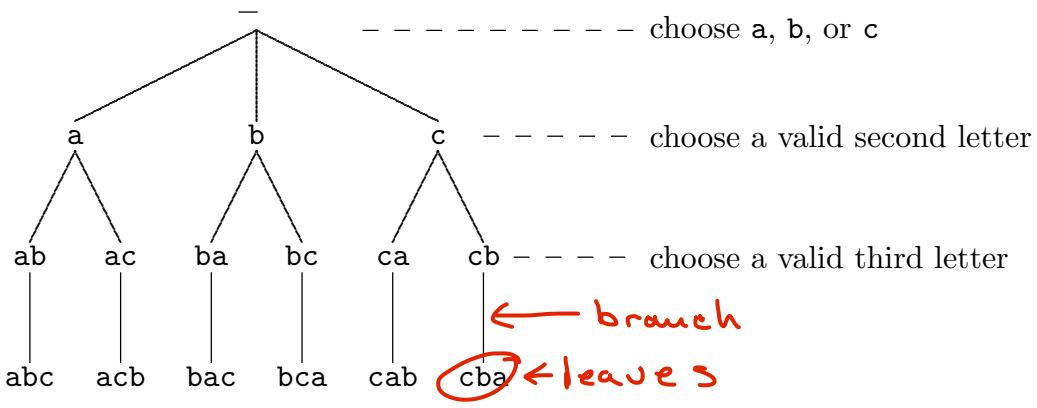
Brute force is the idea of considering all possibilities in a given situation. To do this, at least conceptually, we can build a *tree of possibilities.* To implement this, we might actually build a tree, or a collection of nodes, or we might be able to use method calls instead of literally using nodes.

> In order to facilitate being able to implement brute force by method calls instead of literally building the nodes of a tree, we will encourage using nodes that contain enough information, all by themselves, to allow generation of all the children of the node. In other words, we will discourage using information along a branch.

A First Example: Permutations

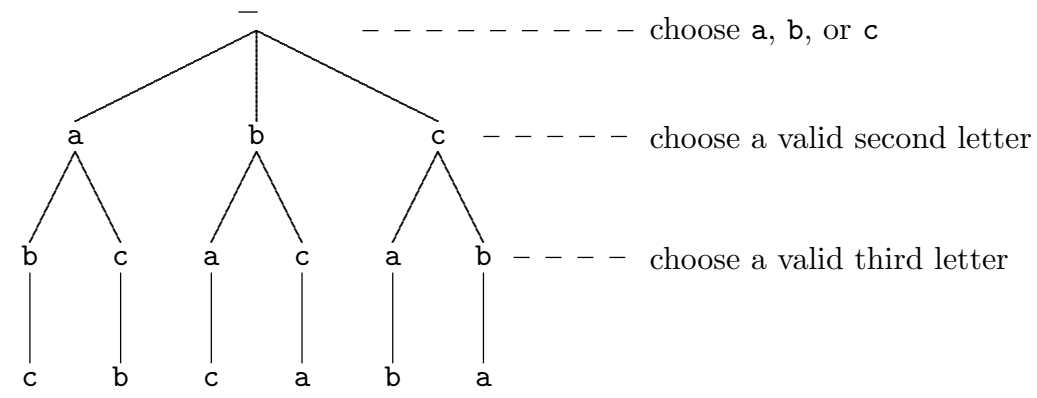
The permutations problem is "given n distinct symbols (for n a positive whole number, of course), make a list of all possible arrangements of those symbols, where each one is used exactly once."

To tackle this problem, say for $n = 3$ (with symbols a, b, and c, say), we simply build this tree, where each level is obtained by asking ourselves "what could we do next from each node?"



The desired list is obtained from the leaves in this tree.

Note that we could have violated our plan of keeping all the information in the nodes by drawing the tree as shown below, where we would have to travel along each branch to determine the desired permutation for each leaf—but this is kind of a bad idea for us (though sometimes it's fine—I certainly don't want to insult any mathy people).



Implementing the Tree of Possibilities

We could implement this process literally, building nodes and so on, but once we understand the conceptual tree, we can sometimes create a recursive method that produces the desired list without the bother—and the possible high amount of memory needed—of actually building the tree.

Example of Brute Force by a Recursive Method Calls

⇒ Let's write a class named `Perms.java` that will generate all the permutations of n things using brute force, implemented by a recursive method. Note that each method call will correspond to a node in the tree.

Pruning

Sometimes we can use the idea of "pruning" in creating a possibilities tree. Pruning is the idea that as we build the tree in some way, often we reach nodes at which we should simply stop, because either they have already violated some rule, or because we realize that no valid possibility can result from continuing from them.

In a way we have already done pruning. Consider the first problem of finding all permutations. We could have approached this by generating *all* words of length n using n symbols, and then pruned all nodes that have duplicate letters. For example, on the second level of the tree, we would produce `aa`, and prune it because any node below it that has n symbols will of course be invalid.

Pruning is sometimes very useful for making brute force more efficient (less work). We will see it in more interesting ways as we go on in the course.

The 0-1 Knapsack Problem by Brute Force

As another example of brute force, let's tackle the 0-1 knapsack problem.

This is a famous problem that we will approach in various ways through the course. It is also known as the "discrete knapsack problem." This problem assumes that we have a collection of items, each with a profit and a weight, and we have a knapsack (a fancy sort of sack, I suppose), and we want to put some of the items in the knapsack, carry it somewhere, and sell the items. So, we want to make the most profit possible, the profit being the total profits of the items in the sack. The catch, of course, is that the sack has a maximum weight capacity.

Here is an example instance, where w_j is the weight of item j, p_j is the profit from selling item j, and the knapsack weight capacity is $W = 12$:

Sorted by profitability

Item j	p_j	w_j	
* 1	100	4	= $25 per lb.
* 2	120	5	= $24
3	88	4	= $22
4	80	4	= $20
* 5	54	3	= $18
6	80	5	= $16

weight

12 lbs

use optimal

Greedy approach doesn't work because the alg. w/ choose 1st 3 choices

The first step in building the tree of possibilities is to figure out what information should be in a node of the tree. It is pretty obvious that we should have the list of items being used stored in the node. It is less obvious, but helpful—especially for pruning—to also store the profit achieved by those items and the weight of those items in the node.

Next we have to figure out how to generate the nodes that are the children of a given node.

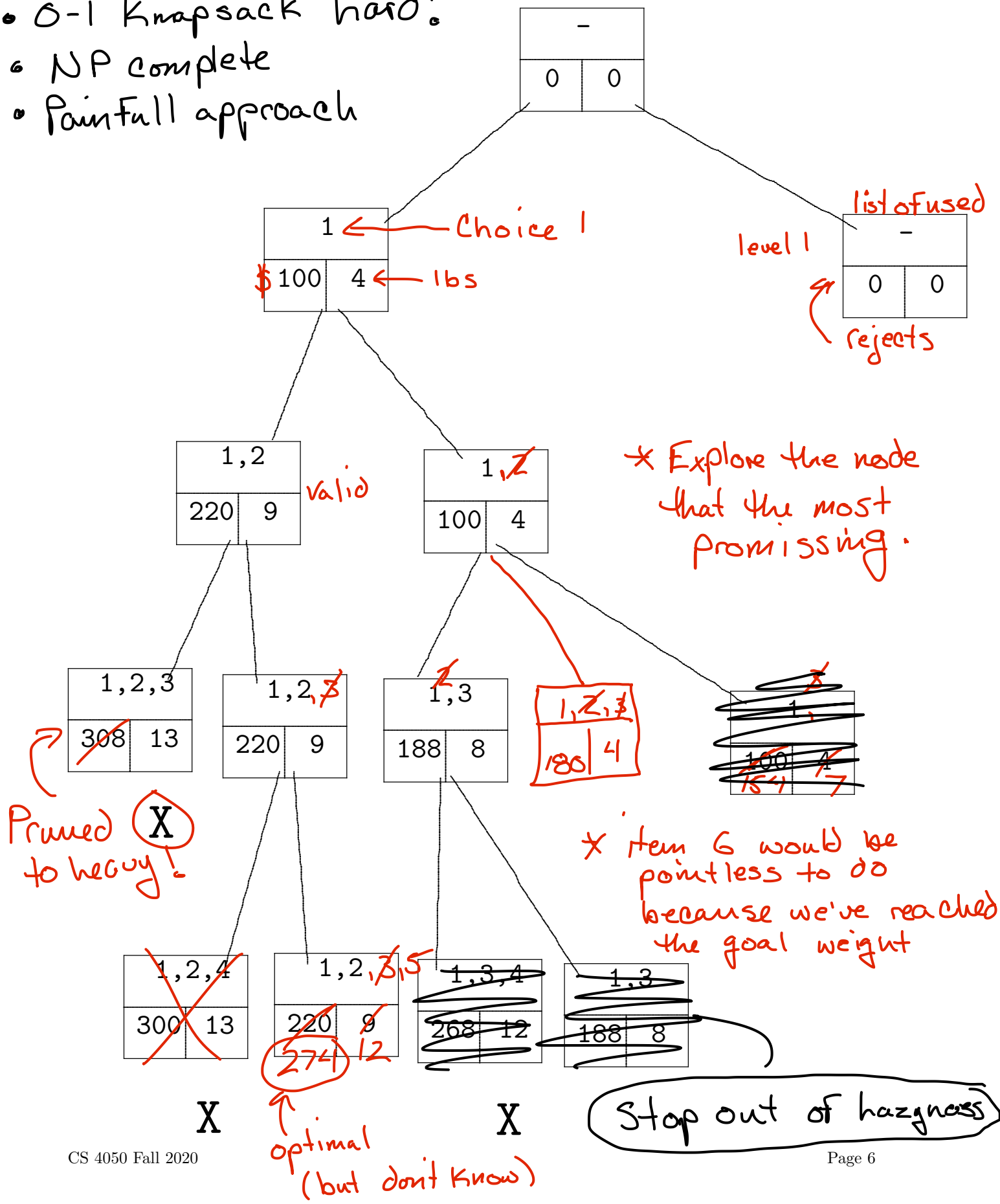
One way to do this would be to pretty much follow the idea for the permutations tree, starting with a root node that has no items, and then saying, well, from there we could decide to use item 1, or item 2, and so on, up through item n, and then repeat this for each node.

It turns out this approach is not what we want, although it would work (later we will want the upcoming approach). So, we instead decide to only give the root node two children, the left one (it's arbitrary which side we use) deciding to use item 1, and the right one saying to never use item 1.

Following this design, we can create the entire tree of possibilities, with the important observation that when a node has items that equal or exceed the sack capacity, we can prune all the nodes below it.

- 0-1 Knapsack hard!
- NP complete
- Painfull approach

Binary Tree



⇒ Let's finish this example.

Note: throughout the course, the exercises will give you practice for the corresponding routine test questions. Some exercises will just be suggested, while others will be turned in for points.

It would be great if you could work on the Suggested Exercises, especially in small groups using Teams or something similar. Suggested Exercises are not included in the numbering scheme, but have titles you can use to talk to others about them.

Of course, you for sure want to do the Required Exercises, because the cumulative points from all the Required Exercises will count for a big part of your course grade.

Finally, note that often more routine exercises are worth more points than harder exercises. The harder exercises that are worth fewer points are intended to help differentiate between C's, B's, and A's. Obviously you should work on the more valuable exercises first, and complete the less valuable ones as time and interest allow.

Suggested Exercise (Combinations by Brute Force)

Consider the combinations problem: given integers $n > 0$ and $k \geq 0$, with $n >= k$, list all the ways to choose k things out of n (use n distinct symbols). Assume that order doesn't matter, in the sense that *ab* and *ba* are considered the same, so it's easiest to use words that have the letters in alphabetical order, like *abc* instead of *acb* or any of the other permutations of those three symbols.

For example, with $n = 4$ and $k = 2$, the desired list is
ab, ac, ad, bc, bd, cd

Design nodes and decisions to build a tree of possibilities for the instance with $n = 5$ and $k = 3$.

Exercise 1 [10 points] (Traveling Salesperson Problem by Brute Force)

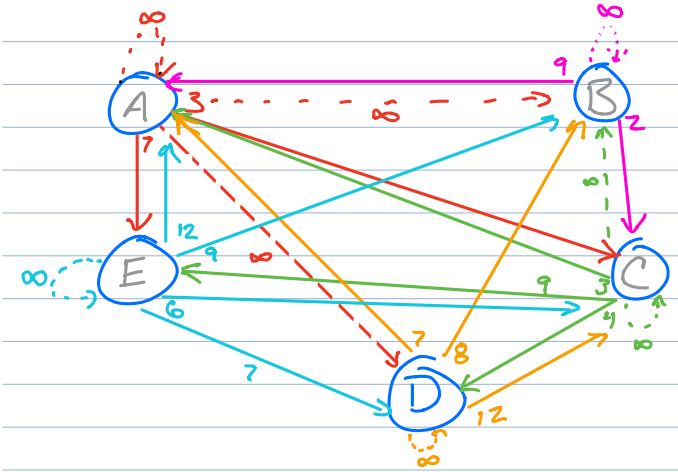
Suppose we have a directed graph (like below) with n vertices labeled 1 through n, implemented using an n by n matrix where row j, column k contains the weight of the edge going from vertex j to vertex k, or a dash if there is no edge from j to k.

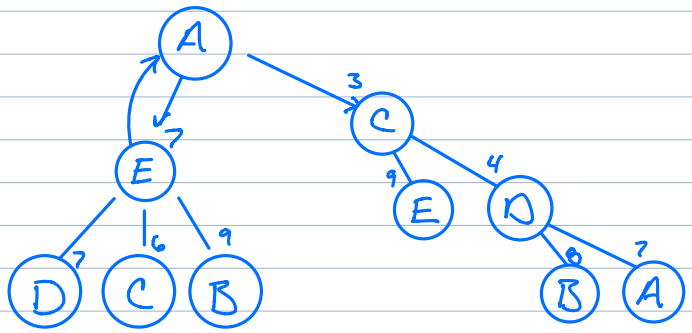
The Traveling Salesperson Problem (TSP) is as follows: given a weight matrix as input, determine the cycle—a path that starts at vertex 1, hits all the vertices exactly once, and ends up back at vertex 1—that has smallest total weight of its edge. Note that the minimum weight cycle might be ∞.

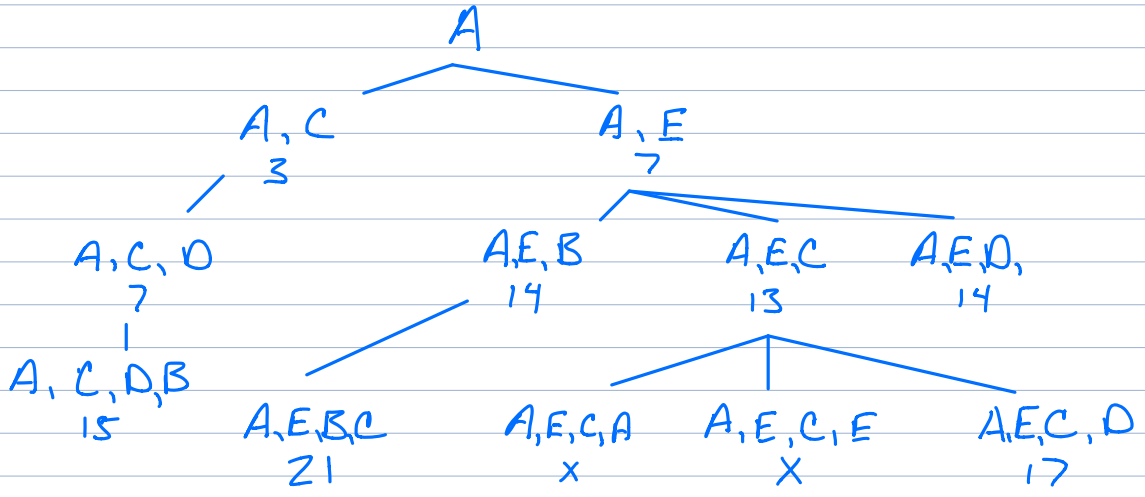
Figure out how to use the brute force technique to solve an instance of this problem, and demonstrate your algorithm on the matrix below. Be sure to think about pruning, and what information should be kept in each node.

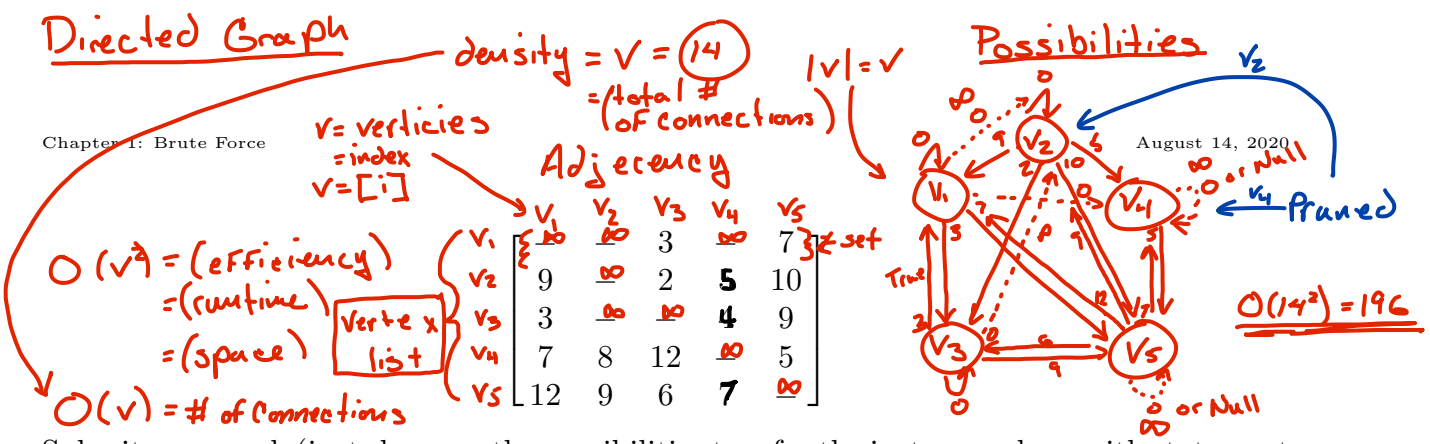
Exercise 1

		A	B	C	D	E
		1	2	3	4	5
A	1	—	—	3	—	7
B	2	9	—	2	5	10
C	3	3	—	—	4	9
D	4	7	8	12	—	5
E	5	12	9	6	7	—









Submit your work (just show me the possibilities tree for the instance, along with statement of the answer, which is the best cycle) through Canvas.

Exercise 2 [4 points] (Sums to n Problem by Brute Force)

Consider the "sums to n problem:" given a positive integer n, list all the different ways to get a collection of positive integers adding up to n. Assume that we don't care about order, so $1 + 2$ and $2 + 1$ are the same possibility.

For $n = 3$, the possibilities are
$1 + 1 + 1$, $1 + 2$, 3

Your job on this project is to create a Java class that will take n as input and produce a list of all the possibilities (with each item in the list on its own row, with the integers in it in nondecreasing order left to right).

Name the class `SumsToN`. Submit your work through Canvas—I need the source code for all the classes in your application (probably just one).

Note that I want to be able to take the source file(s) you send me, put them in a folder, go into that folder and at the command line type

```
javac SumsToN.java
```

and then type

```
java SumsToN
```

and have your application ask, at the command prompt, for the value of n, and then print out the list of possibilities—one per line—in the command window.
