The Brute Force Algorithm Design Technique

You should read the following introductory material, up through page 6, before Session 2

Perhaps motivated by our earlier exploration of some puzzles, we now want to introduce our first and most generally useable design technique: brute force! After this introduction, we will at least consider using brute force for every problem we encounter in the rest of the course (and in all our future work?).

There are at least two reasons for doing this. First, 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 more clever approach. Second, 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. And, brute force is often fairly easy to implement.

But, and this is a crucial point, brute force tends to be hopelessly inefficient, and often there are more clever algorithms that are more efficient. Plus, the more clever algorithms are sometimes actually easier to implement.

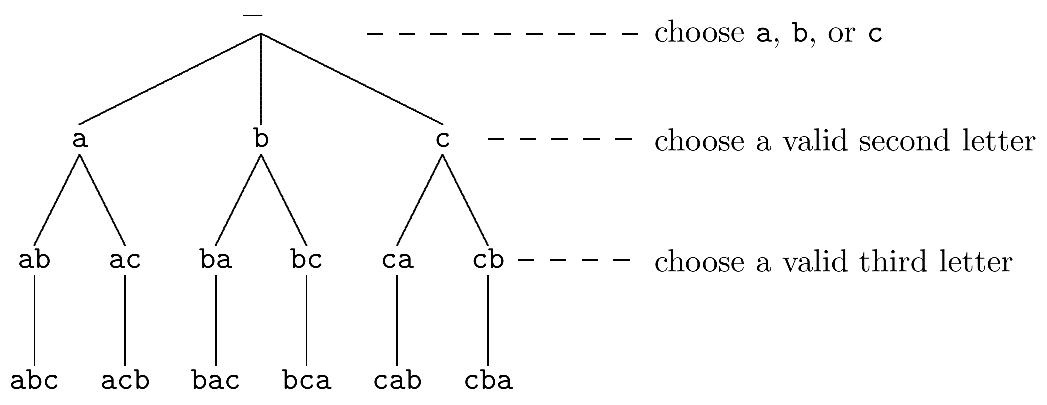
Brute force is the idea of considering all possibilities in a given situation. To do this, at least conceptually, we can build a *tree of possibilities.* To implement this, we might actually build a tree, or we might be able to use method calls instead of literally using nodes.

> 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

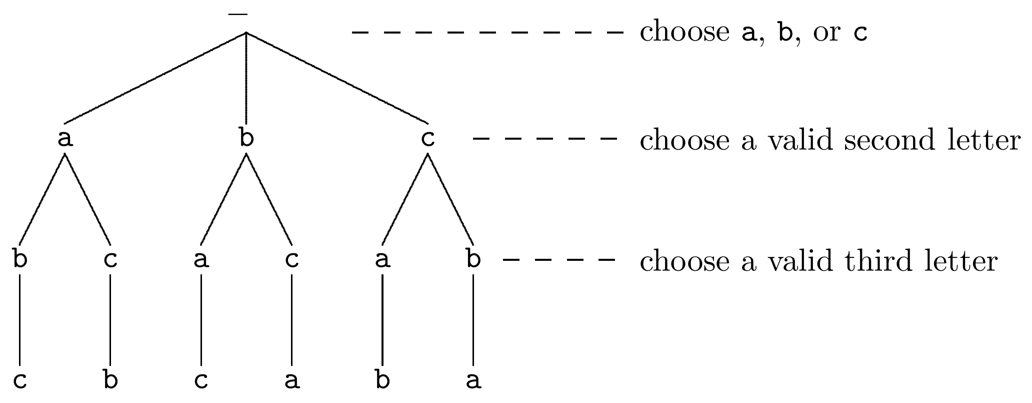
The permutations problem is "given n distinct symbols (for n a positive whole number, of course), make a list of all possible arrangements of those symbols, where each one is used exactly once."

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.



⇒ Embedded Exercise 1

Solve the permutations problem for $n = 4$ in the same way, being sure to keep all the info in the nodes. You will probably want to turn a standard piece of paper sideways and spread out your nodes, in order to allow the tree to fit.

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.

⇒ Embedded Exercise 2

At the course web site, in the folder `Code`, find the file `Perms.java`. Download this file, study the code, and do some sample runs, say for $n = 3$, $n = 4$, and $n = 10$. Note that the code is based very directly on the conceptual possibilities tree—each method call passes in as arguments all the information in the conceptual node.

⇒ Embedded Exercise 3

Consider the problem of forming all different words of length k from n symbols.

For example, for $n = 3$ and $k = 2$, the words are
aa, ab, ac, ba, bb, bc, ca, cb, cc

Draw the possibilities tree for $n = 4$ and $k = 2$.

Pruning

It hasn't shown up in these relatively simple possibilities trees, but a very important idea in such trees is *pruning*. 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.

Pruning is 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.

Introductory Work on Brute Force

In this chapter you will do some hands-on work with the brute force design technique, using the "tree of possibilities" idea. Later, pretty much throughout the semester, we will consider brute force approaches to many problems we encounter.

⇒ **Project 1** [**routine**] 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, so it's easiest to use words that have the letters in alphabetical order, like *abc* instead of *acb*.

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

Draw a tree of possibilities for the instance with $n = 5$ and $k = 3$.

⇒ **Project 2** [**routine**] Hamiltonian Circuit by Brute Force

Suppose we have a directed graph with n vertices labeled 1 through n, implemented using an n by n matrix where row j, column k contains `true` or `false` (or any other convenient notation) and says whether there is an edge in the graph from vertex j to vertex k.

The Hamiltonian circuit problem is to determine whether we can start at vertex 1 and travel along edges of the graph, ending up back at vertex 1 after hitting each other vertex exactly once (and to say the order of vertices in such a circuit, if there is one).

Figure out how to use the brute force technique to solve an instance of this problem, and demonstrate your algorithm on this matrix (where 0 means no edge and 1 means there is an edge):

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

⇒ **Project 3** [**routine**] The 0-1 Knapsack Problem by Brute Force

A famous problem that we will tackle in various ways through the course is known as the "0-1 knapsack problem" or 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 $W = 12$:

Item j	p_j	w_j
1	100	4
2	120	5
3	88	4
4	80	4
5	54	3
6	80	5

Figure out how to create a tree of possibilities for this instance, draw it, and use it to solve this instance of the 0-1 knapsack problem. Think about pruning, to save work, and store data in each node to make pruning, and figuring the child nodes, as comfortable as possible.

⇒ Project 4 [**optional**] 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 by email to `shultzj@msudenver.edu` with all the `.java` files (should not be very many!) needed for your application attached, and all group members (0 through 2) other than the sender CC'ed.

Note that I want to be able to download your attached source files into 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.
