Brute Force

Many standard problems for which a computer is useful involve searching through some space for the best answer to some query, or listing out each item in some sample space. A standard way to view this problems is to write a recursive function that takes in a partial solution (the first k items of a possible solution) filled in, and return/go through all possible ways to complete that partial solution.

Consider a maze where at each turn you have 3 options and each path out is exactly 4 steps. We could enumerate each of the paths out as a sequence of 4 steps, each of which is 0, 1 or 2. An example of one path out would be:

2, 0, 1, 1

Consider the problem of printing out each of these possible $3^4 = 81$ paths out. Recursively, we would formulate the problem a little differently. Given a partial solution (ie. the first two steps), list out all the possible steps that build off of that partial solution. So, imagine listing all paths out that start with 2, 0. Our results ought to be:

2, 0, 0, 0
2, 0, 0, 1
2, 0, 0, 2
2, 0, 1, 0
2, 0, 1, 1
2, 0, 1, 2
2, 0, 2, 0
2, 0, 2, 1
2, 0, 2, 2

In essence, if we want to build all solutions that start with 2, 0, (k = 2 fixed items) what we need to do is try each possible item in index 2, and print out each of those partial solutions separately. Thus, we want to first print all solutions that start with

2, 0, 0 then all that start with

2, 0, 1 followed by all that start with

2, 0, 2

The collection of these solutions comprise ALL solutions that start with 2, 0!

This general technique - using recursion to go through a search space where the recursive function takes in a partial solution and returns/processes all items with that partial solution filled in is a very powerful technique that can be applied to many brute force problems. The framework for a solution in many different cases for these problems share significant commonalities.

Odometer Problem

The first problem we will encounter is the odometer problem. Given the number of digits in an odometer and the number of possible digits for each slot, print out each possible odometer setting. The problem given above is a specific version of the odometer problem with 4 digits, each with 3 possible choices. To code a solution to the odometer problem, we must write in a recursive function that takes in the array that stores the partial solution, as well as an integer k, representing the number of digits that have already been filled in. The job of the recursive function will be to print out each odometer setting with the first k digits from the array fixed as they are. Using the prior example, odometer([2, 0, …, …], 2) should print out the 9 odometer settings previously shown that all start with 2, 0.

Here is the method:

```
public static void odometer(int[] digits, int numDigits, int k, 
boolean printFlag) {

    if (k == digits.length) {
        print(digits, digits.length, printFlag);
        return;
    }

    for (int i=0; i<NUMDIGITS; i++) {
        digits[k] =i;
        odometer(digits, k+1);
    }
}
```

In short, what we do is as follows in the recursive case:

For each possible item that could go in slot k, place it there, and recursively print out all solutions with those first k+1 items fixed.

For the odometer problem, it's very simple to figure out the possible items that could go in slot k. They are ALWAYS the items through NUMDIGITS-1.

In our other problems, this task (determining what is allowed to go in slot k), is more difficult and may require more information to be passed into the method. A relatively slight edit to the odometer problem leads us to a solution to the ubiquitous combination and permutation problems.

Note: for this lecture, I created a general print function that works for each example. The items to be printed come from the array in the first parameter. The number of items printed is the second parameter and the third parameter is the printing mode. More on this after the next couple examples.

Combinations

A combination is a subset of some given set of items. For example, every subset of {0, 1, 2} is as follows:

{}
{0}
{1}
{2}
{0,1}
{0,2}
{1,2}
{0,1,2}

We see that when we are choosing from 3 items, we have 8 possible combinations. In general, there are a total of 2^n combinations that can be formed from choosing from n items. To see this, note that when we form a combination, for each value in the set from which we choose, we have two choices: to put that item into our combination, or not put it in our combination. This set of 2 choices is independent for each item and each set of choices can be combined in any way with the other choices, so to count the total number of combinations, we can simply multiply by 2 for each item that we're choosing from. So, in some sense, we can express a combination as an array of true and false (boolean array) where subset[i] is set to true if item i is in the subset and false otherwise. Using this alternate view, we see that we can also express the 8 subsets listed above as:

{}	F, F, F	0, 0, 0
{0}	T, F, F	1, 0, 0
{1}	F, T, F	0, 1, 0
{2}	F, F, T	0, 0, 1
{0,1}	T, T, F	1, 1, 0
{0,2}	T, F, T	1, 0, 1
{1,2}	F, T, T	0, 1, 1
{0,1,2}	T, T, T	1, 1, 1

For convenience, listed with each subset is a sequence where 1 is substituted for true and 0 is substituted for false. Notice that the items listed on the very left are simply an odometer of 3 digits where the number of possible digits is 2!

Thus, listing out all subsets/combinations of a given set of items is nothing but a specific instance of the odometer problem!!! If we print this information, to make the output look like the items on the left, we need to interpret the 0/1 array. ***If the print flag is set to false, then instead of printing the values in the array, the print function will print each index (in sorted order) that stores a non-zero value.*** For example, if we wanted to use the previous odometer function to print out all subsets of the set {0,1,2,3,4} we would just make the method call:

```
odometer(new int[5], 2, 0, false);
```

Alternate Solution to the Combination Problem

Sometimes we may want combinations of a specific size (exactly k items out of n, for example), or we may want to list each combination as an ordered list of items chosen, in increasing order (instead of a list of 0's and 1's which indicate the membership of each item).

We adapt the odometer solution to solve the problem in this manner by restricting which items get tried in slot k, based on the fact that all of our odometer listings will be in increasing order of item number.

Imagine building all combinations that start with the items 0, 3 (k = 2 fixed items) of items 0 through 9. Since we are listing the combination in increasing order, the possible choices for the slot in index 2 are 4, 5, 6, 7, 8 and 9. (We should NOT try 1 or 2!!!) Notice that the only time we have complete free reign of what to put in slot k is when k is 0. Also, finally notice that ANY partial solution is also a complete solution to the subset problem, so given any partial solution, we should print it!

Here is a solution that prints out all combinations in this format:

```
public static void printcombos(int[] combo, int k) {

    // This itself is a valid combination.
    print(combo, k, true);

    // Determine the smallest item that can go in slot k.
    int start = 0;
    if (k > 0) start = combo[k-1] + 1;

    // Same as odometer, except a different start value.
    for (int i=start; i<combo.length; i++) {
        combo[k] = i;
        printcombos(combo, k+1);
    }
}
```

Now, we will look at a problem that is not identical to the odometer problem, but one that can be solved by making a relatively small change to the solution to the odometer problem, the permutation problem.

Combinations via bitwise operators

For all combinations of n items, there are 2^n possible subsets. As previously noted, we could view an odometer with 2 digits as capable of storing all subsets of a given set of items. Since integers are already stored in binary in the computer, and each bit has two possible values, 0 or 1, we can exploit the bit storage of integers to easily loop through a set of options that represent all possible subsets of a set. (So instead of using an array of booleans or an array of digits that are all 0 or 1, just use a single integer, which, when you peer into it, has 32 0s or 1s stored.)

Here is a list of a few of these subsets, along with the false/true, 0/1, of the same set, where we list our items in reverse (4, then 3, then 2, then 1, then 0):

{0, 2, 3}	FTTFT	01101	13
{0, 1}	FFFTT	00011	3
{3}	FTFFF	01000	8
{1, 4}	TFFTF	10010	18

Notice that the 0/1 representation is just binary, and that if we were to loop through all numbers from 0 to 31, we would cycle through all possible settings of five 0s and 1s, which is the same as cycling through all subsets of values set to right.

In order to exploit this fact, we need to be able to easily obtain each separate bit in an integer. We can do this via bitwise operations. Let's take a quick detour to learn several of the bitwise operators and what they do.

Bitwise Operators

We must be able to easily express a power of two, since the technique above requires us to loop through all non-negative integers upto a power of two minus 1. We can use the left shift operator to do this:

```
1<<n
```

is the number in binary that you obtain when you left shift 1 by n bits. This means placing n 0s to the right of the 1 (moving the 1 to the left by this many bits.) For example,

```
1<<3
```

means `1000` in binary, which is just 2^3 or 8.

More generally, `a<<b` means left-shifting the value of a by b bits. Numerically, unless there is overflow, this is equal to $a2^b$. But, almost always, usually 1 is the number that is left-shifted.

The next bitwise operator is `&`, which is bitwise and. This occurs between two integers. To compute the result, write both out in binary, and in each bit location where both integers have 1s, place a 1, otherwise place a 0.

Here is an example:

```
47 = 101111
29 = 011101
-----------
     001101, so 47&29 = 13
```

Most commonly, the & operator is used to see if a particular bit is on in a number n. Here is the boolean expression which will be true if and only if the kth bit in n is 1:

```
n & (1<<k)
```

or

```
(n & (1<<k)) != 0
```

The former is more succinct, but the latter is probably easier to understand for those used to a programming language other than C/C++.

Basically, the number `(1<<k)` has only one bit set to 1, the kth bit. Thus, when we do a bitwise and, we are guaranteed that all of the other bits in the answer except the kth bit will be set to 0. The kth bit in the answer will be 1 if n's kth is 1 and 0 otherwise. Thus, if this number is 0, the kth bit in n is 0, otherwise it's 1.

The other common bitwise operators are:

```
| - bitwise or
^ - bitwise xor
>> - bitwise right shift
```

A corresponding bit in a bitwise or is 1 if at least one of the input bits is 1.
A corresponding bit in a bitwise xor is 1 if the two bits are different, 0 if they are the same.

So, a bitwise or is good to union two subsets, while a bitwise xor is good at identifying either similarities or differences between two sets, or two binary strings. Here is a Java code segment that prints out all subsets of the set {0, 1, 2, ..., n-1}:

```
for (int mask=0; mask<(1<<n); mask++) {
    for (int i=0; i<n; i++)
        if ((mask & (1<<i)) != 0)
            System.out.print(i+" ");
    System.out.println();
}
```

Permutations

The permutation problem is as follows: Given a list of items, list all the possible orderings of those items.

Generically, we list permutations as all the orderings of the integers from 0 to n-1, inclusive. For example, if we wanted to list all the permutations for n = 3, in lexicographical ordering, these would be:

0, 1, 2
0, 2, 1
1, 0, 2
1, 2, 0
2, 0, 1
2, 1, 0.

There are several different permutation algorithms, but since recursion an emphasis of the course, a recursive algorithm to solve this problem will be presented. (Feel free to come up with an iterative algorithm on your own.)

We utilize recursion as follows, using the following parameters to our recursive function:

1) An array with a partially filled in permutation.
2) A used array, storing which items have already been partially filled in
3) An integer, k, representing how many items have already been filled in.

Technically, one could have all of this information with just item number 1, it makes life easier to pass in items 2 and 3. The job of our function will be to list all permutations of the given partially filled in permutation that have their first k values fixed.

Thus, for example, if k = 1 and our partially filled in array looked like:

2		

Then the goal of the algorithm would be to print out (or process in some way) the following two permutations:

2, 0, 1
2, 1, 0

in that order.

As a second example, imagine the following partially filled in array with k = 4 (for permutations with n = 7):

3	6	0	4			

The algorithm should print out the following permutations:

3, 6, 0, 4, 1, 2, 5
3, 6, 0, 4, 1, 5, 2
3, 6, 0, 4, 2, 1, 5
3, 6, 0, 4, 2, 5, 1
3, 6, 0, 4, 5, 1, 2
3, 6, 0, 4, 5, 2, 1

Recursively, our code ought to do the following:

1) Check if k is equal to n, the length of our permutation array. If so, just print out the fully filled in permutation.

2) If not, iterate through each un-used item, placing it in slot k (in numerical order), and recursively calling the function, noting that now, k+1 items are fixed.

In code, we have the following (assume that the appropriate functions and variables are declared):

```
public static void printperms(int[] perm, boolean[] used, int k)
{
    if (k == perm.length) print(perm, perm.length, true);

    for (int i=0; i<perm.length; i++) {
        if (!used[i]) {
            used[i] = true;
            perm[k] = i;
            printperms(perm, used, k+1);
            used[i] = false;
        }
    }
}
```

The key here is that because we wanted to quickly know whether or not an item was previously used in the partial solution, adding the used array as a parameter to our method greatly facilitated that decision making process in the code. Of course, the key is that when we use that extra parameter, we MUST keep all information consistent. In this case, that meant properly updating the used array whenever any changes were made to the perm array.

Applying Permutation Algorithm to Objects

Say we wanted to go through all the permutations of some array of objects, call it items. Then we can just do the following:

```
public static void printperms(int[] perm, boolean[] used, Type[]
items, int k) {

    if (k == perm.length) process(perm, items);

    int i;
    for (i=0; i<n; i++) {
        if (!used[i]) {
            used[i] = true;
            perm[k] = i;
            printperms(perm, used, k+1);
            used[i] = false;
        }
    }
}
```

The process function we do whatever it needs to do with the items in the following order:

```
items[perm[0]],
items[perm[1]],
items[perm[2]], …,
items[perm[n-1]]
```

Derangements

A derangement is a permutation where perm[i] is NOT equal to i. So, for n = 4, we have the following 9 derangements out of 24 possible permutations:

1, 0, 3, 2
1, 2, 3, 0
1, 3, 0, 2
2, 0, 3, 1
2, 3, 0, 1
2, 3, 1, 0
3, 0, 1, 2
3, 2, 0, 1
3, 2, 1, 0

Luckily, if we think about derangements in terms of code, the process of creating these is very similar to creating permutations. The only difference is that we have a further restriction of what to place in slot k: we can't place the value k in that slot!

Thus, before we place an item in slot k, we must add a check to make sure it is not item k:

```
public static void derange(int[] perm, boolean[] used, int k) {

    if (k == perm.length) print(perm, perm.length, true);

    for (int i=0; i<perm.length; i++) {
        if (!used[i] && i != k) {
            used[i] = true;
            perm[k] = i;
            derange(perm, used, k+1);
            used[i] = false;
        }
    }
}
```

Upwards

Let's consider a final problem, which turns out to be quite similar to the ones previously listed (and very similar to one of them in particular). Define a k-level upward to be a lowercase alphabetic string with letters in sorted order such that the gap between any two adjacent letters in the string contains at least k letters. For example, "act" is a 1-level upward since there is only one letter in between 'a' and 'c', but it *isn't* a 2-level upward. "bit" is an upward of levels 0 through 6, since there are 6 letters in between 'b' and 'i', and "forty" is an upward of levels 0 and 1, since there is only one letter in between 'r' and 't'.

Given a values of n and k, consider the task of printing out all k-level upwards of precisely n letters.

An upward follows the pattern of the combinations which can be listed in increasing order. The trick with listing the combinations was the following lines of code:

```
int start = 0;
if (k > 0) start = combo[k-1] + 1;
```

Namely, instead of trying each letter in slot k, we determined an alternate starting letter for the slot based on the previous letter. In our code for this problem, we will be given a variable skip, that represents the level of the upward, we can use this to modify the code above as follows:

```
char start = 'a';
if (k > 0) start = word[k-1] + skip + 1;
```

Instead of the starting point of our range of possible letters being the very next letter from the last one placed, we want to jump ahead skip more letters, which is precisely what the code above accomplishes! (Note - some other changes have been made to make the code consistent with filling in lower case characters instead of integers starting at 0.)

Here is the method:

```
public static void upwards(char[] word, int skip, int k) {

    if (k == word.length) System.out.println(new String(word));

    else {
        char start = 'a';
        if (k > 0) start = (char)(word[k-1] + skip + 1);

        for (char i=start; i<='z'; i++) {
            word[k] = i;
            upwards(word, skip, k+1);
        }
    }
}
```