TopCoder Training Camp >> Tutorials >> Sets, Subsets, ...

Very often problems deal with some set of "things" and your task is to find a subset of it that is in some sense "optimal". Or you're asked to find the number of "valid" subsets. Or to find the optimal subset among all valid subsets. Or some other more exotic question.

I refer to some TopCoder problems throughout the tutorial and in the exercises. And of course there are also some more general and famous problems. A quote shamelessly from Skiena (see "Further Reading" at the end of the tutorial):

> "Many of the algorithmic problems in this catalog seek the best subset of a group of things: vertex cover seeks the smallest subset of vertices to touch each edge in a graph; knapsack seeks the most profitable subset of items of bounded total size; and set packing seeks the smallest subset of subsets that together cover each item exactly once."

This tutorial mainly covers generation of subsets as exhaustive brute force search (which is good enough for quite a few problems). Optimizations and some more mathematical facts are also discussed.

Contents:

Basics

A "set" is just some collection of "things", for example the set of three people { Anna, Bob, Caesar } or the set of four numbers { 496, 6, 28, 8128 }. The things contained in a set are called "elements" of that set, so 28 is an element of the set { 6, 28, 496 }, whereas 42 is not.

Note that sets are not ordered, i.e. { 23, 42 } = { 42, 23 }. They also don't care about duplicates, i.e. { 6, 6, 28 } = { 6, 28 }. However, I will usually write my example sets without duplicates and often they will look ordered. Note that if the input to a programming problem is a set, you might come across duplicates and the set might not be ordered. Be careful, watch out for that!

The "size" or "cardinality" of a set A is written as #A or |A|. It tells us how many elements are in the set, for example |{6, 28, 6}| = 2. There are finite and infinite sets, meaning they have finitely or infinitely many elements. Even though infinite sets are mathematically nice, they play only a very little (if any) role in TopCoder problems, so I will only discuss finite sets here.

We are programmers and want to work with sets, therefore we have to hold them in some data structures. Even though sets don't offer an idea of order, in your program you have them stored somehow, maybe in an array that contains the elements. Because of this, it's legitimate to talk about the "first element" of a set or the "i-th element". When I say something like that, remember that by this I mean the "first element **in my representation** of the set" or similarly the "i-th".

A "subset" of some set B is a set that contains only elements of B. For example, here are all subsets of the set { 3, 1, 4 }:

{ }, { 3 }, { 1 }, { 4 }, { 3, 1 }, { 3, 4 }, { 1, 4 }, { 3, 1, 4 }

The first set in that list is a special set, called the "empty set". Naturally, the empty set is a subset of any set. The last set is also a bit special, since it's the same as the whole set B. You can see that every set is a subset of itself.

If you have a set of N elements, how many subsets are there? In the above example, N=3 and we had eight subsets. The answer is 2^N, in the example $2^3 = 8$. Why is this? To build our subset, let's start with the first element. We can either include it or leave it out. A choice between two possibilities. Then we go on to the next element and make a choice to include it or not. And so on. For each of the N elements, you have two choices, so there are 2^N possible resulting subsets. Convince yourself that with this method you'll really get each subset exactly once. You will not create a subset twice and every possible subset will be generated. We will also see this again a little bit later.

Since 2^N is a fairly easy formula, you will probably never see a problem that just asks for the number of subsets of a set. But it's good to know, since you might need it as part of a larger task.

Set theory is a large large part of mathematics and far more complicated and richer than what I'll cover here, since here I'm more

interested in basic computational needs. For a start on more, look at MathWorld.

LightMeal Example

Let's look at an example. In SRM 87, the division 1 level 3 problem dealt with a set of food items. For each item, you are given the amount of calories and of three vitamins. The task was to find the best meal that had at least 100% of the daily need of each vitamin. A meal is "better" than another if it has less calories. Look at practice room 112 for the complete problem statement. A short example:

Item number	Calories	Vitamins		
0	28	20	33	98
1	23	69	33	1
2	6	90	33	1
3	42	10	40	1
4	496	100	100	100

Item 4 alone has the needed vitamin amount, but also a huge amount of calories. Can we do better without it? Yes, when we put items 0, 2 and 3 together, we have vitamin sums of (120, 106, 100). And they only have 76 calories together. There's no better way and thus the result is 76 (the problem asked for the number of calories as result).

A "meal" was in fact just a subset of the set of food items. But here we have a restriction, we're not concerned about *all* subsets, but only about *valid* subsets, those that contain enough vitamins. Of all those valid subsets, we're searching for the one with the fewest calories.

Let me tell you a secret. Since the maximum number of food items was 20, there are at most 2^{20} subsets, which is about a million. And of course there are at most that many *valid* subsets. Well, you can do a lot* in the eight seconds TopCoder gives you. The easiest way to solve this problem is to generate *all* subsets, determine for each one if it is valid (has enough vitamins) and keep track of the minimum number of calories of all valid subsets. This is in fact fast enough.

(*: try `int j=0; for( int i=0; i<100000000; ++i ) j+=i;`, it will run about a second on the TopCoder server for these 100 million iterations)

So how do we generate all subsets? One of the easiest and most intuitive ways is to use a recursive function, which works just like I explained in my argument for the 2^N subsets. Here's a general outline of the algorithm:

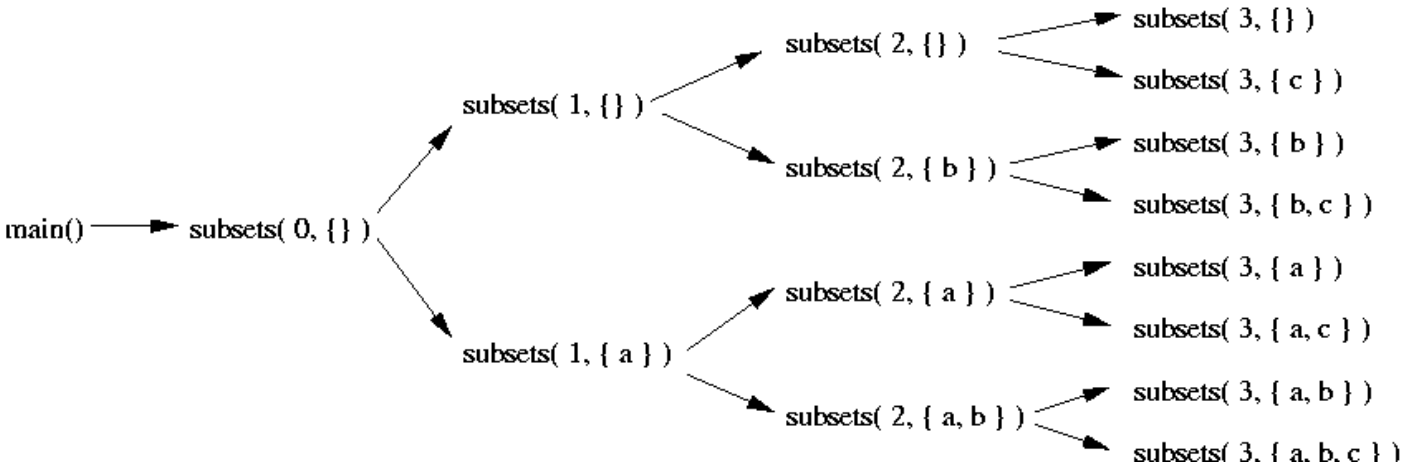
```
main () {
    subsets( 0, {} );
}

subsets ( int i, Subset s ) {
    if( i == N ){
        doSomethingWithTheBuiltSubset( s );
        return;
    }
    subsets( i+1, s );
    subsets( i+1, s + element(i) );
}
```

The function "subsets(i, s)" will make all choices for the remaining elements i to N-1 (I assume the elements are numbered from 0 to N-1 where N is the size of the whole set). That's why it's called in main with argument zero, because it still has to decide for all elements. The second argument holds the partially built subset, so of course main starts the building with the empty set.

`subsets` first looks if it has already decided for all elements (i.e. if i==N) and if yes, then we can do something with the built subset. In the LightMeal problem, it would check if it has enough calories and if it's a new record holder for fewest calories.

If we're not yet finished (i.e. if i<N), then `subsets` just tries both possibilities for element i. With the first recursive call, it makes the choice to leave it out, with the second one it includes it in the subset.

The call tree of this with when the whole set is { a, b, c } will look like this:



You can see that at the recursion bottom (at the right), where i==3, each possible subset of { a, b, c } appears exactly once. Now that you got an idea of how this works, let's see it in action and real code that solves the LightMeal problem. Note that I don't explicitly build the subset, but rather just the sum of vitamins and calories, since that's all I need to care about. So instead of adding an element to some subset, I add its values to some sums.

```
struct LightMeal {

  int n, best, cal[20], A[20], B[20], C[20];

  void check_subsets ( int i, int cals, int a, int b, int c ) {

    //--- Made a choice for all items?
    if( i == n ){
      if( a>=100 && b>=100 && c>=100 )
        best = min( best, cals );
      return;
    }

    //-- Either leave this one out or take it in.
    check_subsets( i+1, cals, a, b, c );
    check_subsets( i+1, cals+cal[i], a+A[i], b+B[i], c+C[i] );
  }

  int calories ( vector<string> items ) {

    //--- Transfer the input to arrays for easier and faster access.
    n = items.size();
    for( int i=0; i<n; ++i ){
      istringstream sin( items[i] );
      sin >> cal[i] >> A[i] >> B[i] >> C[i];
    }

    //--- Check all possible subsets of meals.
    best = 20*100+1;
    check_subsets( 0, 0, 0, 0, 0 );

    //--- Answer.
    return best;
  }
};
```

Here the main function is "calories", which first parses the input into some arrays for easier access. Then it calls the recursive "check_subsets" function. The first parameter is the iterator i, just as before. But instead of one subset parameter, we have the four sums we're interested in.

Sometimes it's even easier to use global data structures to build the subset or the values you want to know about it, but that depends on the problem and your preferences and current mood. In the above program, I use global variables for the whole set of items and to keep track of the "best" valid subset. Another possibility would've been to let the function *return* the best answer:

```
struct LightMeal {

  int n, cal[20], A[20], B[20], C[20];

  int check_subsets ( int i, int cals, int a, int b, int c ) {

    //--- Made a choice for all items?
    if( i == n ){
      if( a>=100 && b>=100 && c>=100 )
        return cals;
      else
        return 20*100+1;
```

```
    }

    //-- Either leave this one out or take it in.
    return min( check_subsets( i+1, cals, a, b, c ),
                check_subsets( i+1, cals+cal[i], a+A[i], b+B[i], c+C[i] ));
  }

  int calories ( vector<string> items ) {

    //--- Transfer the input to arrays for easier and faster access.
    n = items.size();
    for( int i=0; i<n; ++i ){
      istringstream sin( items[i] );
      sin >> cal[i] >> A[i] >> B[i] >> C[i];
    }

    //--- Check all possible subsets of meals and return the best found:
    return check_subsets( 0, 0, 0, 0, 0 );
  }
};
```

The Bitfield Counter Method

You have already seen a recursive method to generate all subsets. Now I'll show you an iterative one. I call it "Bitfield Counter", but it's also known under other similar names.

The idea is to exploit the fact that a subset either includes or leaves out each element. Two choices, like each bit of a binary number. One representation of a subset of a set of size N is to have a bitfield of N bits. A '1' in bit i tells us that element i is included in the subset, a '0' means it's left out. For example, if we have the set { 9, 16, 25 }, the subset represented by the 3-bit bitfield "101" is the set { 9, 25 }.

Here's a complete list of subsets and the bitfields that represent them, if the set we're talking about is { 9, 16, 25 }. The spacing is intentionally weird, to show the connection between the bitfield and the represented subset.

Bitfield	Subset represented by the bitfield
000	{ }
001	{ 25 }
010	{ 16 }
011	{ 16, 25 }
100	{ 9 }
101	{ 9, 25 }
110	{ 9, 16 }
111	{ 9, 16, 25 }

You can probably already guess how we generate all the subsets now. We just generate all bitfields. And what could be easier than to just use an "int" variable, declare the lower N bits as our bitfield, and then count from 0 to 2^N-1 with that variable? Again, here's some outline:

```
for( int B=0; B<(1<<N); ++B ){
    clearSubset();
    for( int i=0; i<N; ++i )
        if( (B & (1<<i)) > 0 )
            addToSubset( element(i) );
    useBuiltSubset();
}
```

Here B is the variable that holds our bitfield representation of the subset. The outer loop lets B run through all possible subset representations. Inside the loop, we first clear the subset and then extract from B which elements are included in the subset it currently represents. We do this with a second loop that uses bitshifting and bitwise "and" to test for each bit i if it's set to '1' in B, and if yes, we add element i to the current subset.

The above code only works for N<31, since 2^{31} is not an "int" anymore, it's too big. We could use "long long" (that's "long" in Java) variables, because they offer more bits. But think about it. The loop runs for 2^N iterations and with N=31 this will be more than 2 billion times. That will time out anyway, even if you do nothing inside the loop. Conclusion: The N<31 constraint for this method is irrelevant, for higher limits we need a totally different approach anyway.

Enough smalltalk, let's see how this looks like when we apply it to the LightMeal problem:

```
struct LightMeal {

  int calories ( vector<string> items ) {
```

```
    //--- Transfer the input to arrays for easier and faster access.
    int n = items.size();
    int value[20][4];
    for( int i=0; i<n; ++i ){
      istringstream sin( items[i] );
      for( int j=0; j<4; ++j )
        sin >> value[i][j];
    }

    //--- Check all possible subsets of meals.
    int best = 20*100+1;
    for( int B=0; B<(1<<n); ++B ){
      int sum[4] = {};
      for( int b=0; b<n; ++b )
        if( B & (1<<b) )
          for( int j=0; j<4; ++j )
            sum[j] += value[b][j];
      if( (sum[1] >= 100) && (sum[2] >= 100) && (sum[3] >= 100) )
        best = min( best, sum[0] );
    }

    //--- Answer.
    return best;
  }
};
```

Like in the recursive variant, I first parse the input into arrays. Then I generate all subsets and test each one, keeping track of the best found answer so far. I used "B" for the bitfield and "b" to extract the items from the set represented by B. As before, I don't explicitly build the subset, but only the sums of calories and vitamins that it contains. That's all I need. Compare this program to the outline code above, you'll easily identify the clearing of the subset, the adding of elements and the using of the finished subset.

Optimization by Breaking Subtrees

Very often you are in the middle of building a subset and you can see that it's of no use to fill it with more elements. For example, in the LightMeal problem, once the partial subset of items has enough of all vitamins, why would you add more items? The subset is already valid and with more elements you would add more calories, which is of no use. Therefore you can break early:

```
void check_subsets ( int i, int cals, int a, int b, int c ) {

  //--- Already enough vitamins?
  if( a>=100 && b>=100 && c>=100 ){
    best = min( best, cals );
    return;
  }

  //--- Made a choice for all items?
  if( i == n )
    return;

  //-- Either leave this one out or take it in.
  check_subsets( i+1, cals, a, b, c );
  check_subsets( i+1, cals+cal[i], a+A[i], b+B[i], c+C[i] );
}
```

In some problems, this can save you a considerable amount of time, since you cut off entire subtrees of the call tree (remember the picture from the beginning of this tutorial). Unfortunately, in the LightMeal problem it will save us nothing in the worst case. And the worst case is the one you have to fear. TopCoder doesn't care if you average less than 8 seconds. Exceed it once, for a single testcase, and you're out. In LightMeal, an input that has 20 items of "5 5 5 5" will make us generate all subsets again. Fortunately N=20 is small enough so that we don't even need any optimizations for this particular problem.

The iterative bitfield counter method can also profit from similar optimizations. However, since we build the subsets separately instead of with a tree, the benefits are very small. You just can't cut off entire subtrees when there aren't any.

There are several more sophisticated ideas that can speed up your execution time and I want to mention a few that came to my still troubled mind:

- If you don't have enough elements in your subset yet, but you know for some other reason that it's of no use to go on, you can also break. For LightMeal, we can break if the sum of calories in the current partial subset already exceeds the best found answer so far.
- Sort the elements. With most algorithms, the first element will not be included in the first half of the subsets, but in all subsets of the second half. So if it's unlikely that you want this element (maybe it has a horrific amount of calories and almost no vitamins?) then the first place is a good place for it. If it's an element you'd like to include (lots of vitamins, almost no calories), it should be among the last elements.

Another reason is that among the last elements, the volatility (ha! TC taught me a new word) is much higher than among the first few. Naturally we'd like to experiment with the most interesting elements the most, so let's place them at the end. For LightMeal, sort them by vitaminsSum/(calories+1) (don't divide by calories alone, could be zero). Or by vitaminsSum-calories. Any heuristic that will reward vitamins and punish calories.

This reordering alone usually buys you nothing. But combine it with the breaking mentioned before and you might be able to break more often and earlier. In LightMeal, you might find low "best" values early and then be able to cut off larger subtrees.

- Look in your favourite "introduction to algorithms" book under "branch and bound" to find other techniques.

But before you write long code to implement complicated optimizations, think about worst cases. Do the optimizations you have in mind really save your butt in the worst case? Believe me, either a mean competitor or the system tests will provide a worst case input. I promise. This is not real life and we don't care about average time.

Subsets of Specific Size

Sometimes we're not interested in all subsets, but only in those of a certain size K, called "k-subsets". Two examples:

- Big2 (practice room 125, problem 3) - We are given a set of 13 cards and have to consider all subsets containing exactly 5 cards, that is, all 5-subsets.
- Hiring (practice room 125, problem 2) - We are given up to 50 people and have to find the best valid subset that contains exactly 3 people.

In both problems, we have to inspect all subsets of size K (5 for Big2 and 3 for Hiring) of a set of size N (13 for Big2, up to 50 for Hiring). Let's first find out how many there are, i.e. in how many different ways can we choose K elements out of N? This is written as "n choose k" or shorter as "nCk". I used lowercase letters for N and K there, because that's how it's usually written. I have to admit that I use uppercase N and K throughout this tutorial just because they're more visible in the text.

A small example with the set {a,b,c,d,e,f,g,h} (N=8) and K=3: We begin by picking one out of the 8 elements, say 'c'. Then we pick another one out of the remaining 7 elements, say 'g'. Then we pick a third element out of the remaining 6, say 'd'. So overall we have 8*7*6 possible outcomes, right? After all, we had first 8, then 7, then 6 possibilities. In fact, this is true.

Unfortunately, we just built the set {c,g,d} and with other choices we might have built the set {g,d,c}. Sets are not ordered and thus these two are the same, you say? You're damn right. With the above procedure, there are six possible ways to get to each set, for example the set {c,d,g} can be built with these choices:

cdg / cgd / dcg / dgc / gcd / gdc

Note that these are just the six permutations of these three elements. Seems like we have to divide our number of possible outcomes by six, and then we arrive at:

$$8 \text{ choose } 3 = \frac{8 * 7 * 6}{1 * 2 * 3}$$

Another example that also illustrates that (n choose k) = (n choose (n-k)):

$$8 \text{ choose } 5 = \frac{8 * 7 * 6 * 5 * 4}{1 * 2 * 3 * 4 * 5}$$

Do you see how beautiful the two 4's and the two 5's cancel each other out? I won't go into much more detail about the choose function here. There are several ways to compute it and I could fill another tutorial just on this issue. In fact, I probably will. But here I'll just show you one simple way that computes the fraction left to right:

```
int choose ( int n, int k ) {
  k = Math.min( k, n-k );
  if( k < 0 )
    return 0;
  int result = 1;
  for( int i=1; i<=k; ++i )
    result = result * (n-i+1) / i;
  return result;
}
```

It starts with 1, then goes to 8/1, then to (8*7)/(1*2), then (8*7*6)/(1*2*3), for example. How can I be sure that the intermediate values are integers? Well, the intermediate values are all "choose numbers" themselves, (8*7)/(1*2) for example is 8C2, and since that represent the "number of ways" we can do something, it should be an integer. That's no good proof, but I hope you'll be convinced for now.

You will also most likely find the choose function on your pocket calculator, look out for "nCk" or "nCr". Now that we know a little bit about the number of k-subsets of a set of size N, let's apply this knowledge to the problems Big2 and Hiring. In Big2, N and K are both fixed, so we always have 13C5 = 1287 subsets. That's nothing, an easy brute force approach that generates and tests them all will be sufficient. In Hiring, K is fixed and N is in the worst case 50, so we will never have more than 50C3 = 19,600 subsets. Again, that can easily be done with an exhaustive search.

I will step away from Big2 and Hiring now (because I want to have small example programs here and because I want to make you solve Big2 and Hiring as exercises) and solve the following artificial problem in different ways: We're given a set of N names. For each subset of K names, print the initials of the people in the subset. This will become clear in a moment, when you see the first program and its output.

The first solution uses some for-loops to pick the elements, altogether K loops. Play a little bit with it, add or remove some names.

```
public class KSubsetsLoop {

  public static void main ( String[] args ) {
    new KSubsetsLoop();
  }

  KSubsetsLoop () {
    //--- Initialize the data we're using.
    String[] set = new String[] { "Anna", "Bob", "Carla", "Dennis" };
    int n = set.length;

    //--- Now let's have some fun!
    for( int i=0; i<n; ++i )
      for( int j=i+1; j<n; ++j )
        System.out.print( "" + set[i].charAt(0) + set[j].charAt(0) + " " );
    System.out.println();
  }
}
```

This neat program has the following output (quotes for clarity only): "**AB AC AD BC BD CD** ". The outermost loop "picks" the first element, the next inner loop picks the second element, and so on (if we had more loops). In order not to produce duplicates (remember the discussion above), a loop picks only from the elements with higher index than the previously picked one. That's why the second loop starts with j=i+1. Try adding one or two loops to the above program to build 3-subsets and 4-subsets.

Looks fine, right? Yes, it does, and for the Big2 and Hiring problems it works very well, since K is fixed (5 and 3, respectively). But what if you don't know K in advance, what if it's part of the input? You can't add or remove for-loops from your program at runtime unless you're a really weird hacker. Instead, we will add the for-loops in a recursive fashion:

```
public class KSubsetsRecursiveLoop {

  public static void main ( String[] args ) {
    new KSubsetsRecursiveLoop();
  }

  KSubsetsRecursiveLoop () {
    //--- Initialize the data we're using.
    set = new String[] { "Anna", "Bob", "Carla", "Dennis" };
    n = set.length;
    k = 2;

    //--- Now let's have some fun!
    buildSubsets( 0, 0, "" );
    System.out.println();
  }

  //---------------------------------------------------------------------

  int n, k;
  String[] set;

  //---------------------------------------------------------------------

  void buildSubsets ( int i, int j, String subset ) {
    //--- Is the subset complete?
    if( j == k ){
      System.out.print( subset + " " );
      return;
    }
    //--- Ok, let's add more.
    for( ; i<n; ++i )
      buildSubsets( i+1, j+1, subset+set[i].charAt(0) );
  }
}
```

That program has the same output as the previous one. The first parameter i points to the next element to decide about. It is increased by

1 in the recursive call, just like it is from one loop to the next in the first method. The second parameter j counts how many elements were already included in the subset. Now play around with this one, too. Add/remove names, increase/decrease K. Maybe read the input from the console, so that the user can pick a value for K. See how easy this program adaps to a change of K? So if you have a fixed K than you might want to use for-loops, it's easy to code. But you can also use the recursive solution for fixed K, of course. For variable K, you can't use the first method.

Mixing and Order

In this section, we'll see that the algorithms I presented you so far for building subsets and k-subsets, are very closely related. We will also see the subsets generated in a different order. Let's start by using the bitfield counter method for producing k-subsets:

```
public class KSubsetsBitfieldCounter {
  public static void main ( String[] args ) {

    //--- Initialize the data we're using.
    String[] set = { "Anna", "Bob", "Carla", "Dennis" };
    int n = set.length;
    int k = 2;

    //--- Now let's have some fun!
    for( int B=0; B<(1<<n); ++B ){
      if( bitcount(B) != k )
        continue;
      String subset = "";
      for( int i=0; i<n; ++i )
        if( (B & (1<<i)) > 0 )
          subset += set[i].charAt(0);
      System.out.print( subset + " " );
    }
    System.out.println();
  }

  static int bitcount ( int n ) {
    return (n==0) ? 0 : (n&1) + bitcount( n/2 );
  }
}
```

It works by generating the bitfield representation for *all* subsets and then only processes those with K bits set to 1, that is, those that represent subsets of size K. This solution prints "**AB AC BC AD BD CD** ". That's not too bad, all subsets are there. But the order changed, for example "AD" used to be generated before "BC". Still, each subset in itself is ordered, for example, 'A' comes before 'D' in "AD".

In fact, all subsets built by any algorithm in this subset are ordered in itself. Did you ever wonder why? It's because I always had the original set ordered, for example "Anna" came first, "Dennis" came last. And it's also because all algorithms here make the decisions left-to-right. They first decide for element 0, then for element 1, and so on.

Now why does the previous program print "BC" before it prints "AD"? Well, consider the bitfields that represent them: 0110 for "BC" and "1001" for "AD". Note that we placed "Anna" at the "leftmost" index 0, but the *bit* 0 is the rightmost. In most problems, the order in which the subsets are generated, doesn't matter. After all, often you look at *all* of them anyway. But what when we insist on getting the old order with the bitfield counter? Let's analyze what we have and what we want by writing it down (often a good idea):

The bitfields which represent subsets of size 2	What it currently represents	What we want it to represent
0011	AB..	AB..
0101	A.C.	A.C.
0110	.BC.	A..D
1001	A..D	.BC.
1010	.B.D	.B.D
1100	..CD	..CD

(This table is btw exactly what I did to get it right, had some trouble before.) Hmm, what we want seems to coincide with the zeros in the bitfield, not looking at the bitfield reversed. Maybe if we let the i-th element of our set be represented by the (n-1-i)-th bit instead of the i-th bit and if we switch the roles of zero and one-bits we'd get it? Yes, and the following program works just fine:

```
public class KSubsetsBitfieldCounterLexicographic {
  public static void main ( String[] args ) {

    //--- Initialize the data we're using.
    String[] set = { "Anna", "Bob", "Carla", "Dennis" };
    int n = set.length;
    int k = 2;

    //--- Now let's have some fun!
    for( int B=0; B<(1<<n); ++B ){
      if( bitcount(B) != k )
```

```
        continue;
      String subset = "";
      for( int i=n-1; i>=0; --i )
        if( (B & (1<<i)) == 0 )
          subset += set[n-1-i].charAt(0);
      System.out.print( subset + " " );
    }
    System.out.println();
  }

  static int bitcount ( int n ) {
    return (n==0) ? 0 : (n&1) + bitcount( n/2 );
  }
}
```

Before we go on to the next method, let's analyze the above one a bit. Don't you have a bad feeling about looping through *all* subsets and the only process a few of them, throwing others away because they have the wrong size? For Big2, we will produce 2^{13}=8192 subsets and then process 13C5=1287 of them. That's ok, no problem. For Hiring, we can use "long long" or "long" variables (because we need 50 bits), and then we would produce 2^{50}>10^{15} subsets and then process only 50C3=19600 of them. Huh, that's a total waste. And btw, 10^{15} iterations will take approximately 130 days on the TopCoder server, slightly more than the 8 seconds we have ;-)

So if you like the bitfield counter method and want to use it for k-subsets, first make sure that you're not wasting too much runtime. I can not say often and loud enough that the very first thing I do when I consider a brute force method is to grab my pocket calculator and compute the worst case costs. Always.

Now let's go on, try something new. Let's generate all k-subsets with our initial recursive method to generate all subsets:

```
public class KSubsetsRecursive {
  public static void main ( String[] args ) {
    new KSubsetsRecursive();
  }
  KSubsetsRecursive () {
    //--- Initialize the data we're using.
    set = new String[] { "Anna", "Bob", "Carla", "Dennis" };
    n = set.length;
    k = 2;

    //--- Now let's have some fun!
    buildSubsets( 0, 0, "" );
    System.out.println();
  }
  //-------------------------------------------------------------------------

  int n, k;
  String[] set;

  //-------------------------------------------------------------------------
  void buildSubsets ( int i, int j, String subset ) {
    //--- Is the subset complete?
    if( j == k ){
      System.out.print( subset + " " );
      return;
    }
    //--- No more elements?
    if( i==n )
      return;

    //--- Ok, let's go on.
    buildSubsets( i+1, j, subset );
    buildSubsets( i+1, j+1, subset+set[i].charAt(0) );
  }
}
```

Oops, that produces the output "**CD BD BC AD AC AB** ". But this can easily be fixed, just change the order of the two recursive calls. The first time include the element, the second time leave it out. Why does this work? Left as an exercise for you ;-) It might help to look at the call tree from the beginning and think about what happens if we change the order of the two recursive calls.

Now to the biggest success story of this tutorial. I asked myself what it would look like if I used my algorithm to generate k-subsets to generate *all* subsets instead. After all, there's only one recursive call, which means less typing and it might look more elegant. Here's the result:

```
public class SubsetsLexicographic {
  public static void main ( String[] args ) {
    new SubsetsLexicographic();
  }
  SubsetsLexicographic () {
    //--- Initialize the data we're using.
    set = new String[] { "Anna", "Bob", "Carla", "Dennis" };
```

```
    n = set.length;

    //--- Now let's have some fun!
    foo( 0, "" );
  }

  String[] set;
  int n;

  void foo ( int i, String subset ) {
    System.out.println( '"' + subset + '"' );
    for( ; i<n; ++i )
      foo( i+1, subset + set[i].charAt(0) );
  }
}
```

Wow, that looks indeed cool. But here's what really amazed me: The output. It is lexicographically ordered. Try to achieve that with our initial recursive algorithm or with the bitfield counter. I dedicate this program to Steven Skiena, who writes

> "Unfortunately, it is surprisingly difficult to generate subsets in lexicographic order. Unless you have a compelling reason to do so, forget about it."

Notice that of course in order to work so nice, the original set has to be ordered, as discussed earlier, but here it's really essential. Here's the output of the program:

```
""
"A"
"AB"
"ABC"
"ABCD"
"ABD"
"AC"
"ACD"
"AD"
"B"
"BC"
"BCD"
"BD"
"C"
"CD"
"D"
```

Should you ever feel the need to produce all subsets in lexicographic order, this might be a good one to start. Maybe it's what the problem asks for. Or you want to find the lexicographically smallest "valid" subset, and with this method you can stop the search as soon as you come across the *first* valid subset. However, I doubt that it is very useful for TopCoder, but for the ACM programming contest, it might be valuable. And btw, did I mention that this program has pretty damn short code? I think I'll try it next time I attack a subset problem with brute force.

Multisets

I pretty much know what I want to include in this section, but it will take me some time and it's fairly independent from the rest, so I decided to make the tutorial public without multisets and I'll include them soon.

Efficiency Guidelines

Generating all subsets or all subsets of a certain size is a brute force approach, and even though it works for many problems, there are some where it's just too slow. Here are some guidelines how to determine feasability of this approach:

- If N=20, generating all subsets takes no time. Do it.
- If N=30, generating all subsets is too slow. Sometimes you can still do it recursively with good enough breaking conditions. But generally it might be wise to look for an approach that doesn't generate all subsets. Maybe there's a greedy algorithm or you can use dynamic programming?
- Somewhere between N=20 and N=30 is the border between feasible and infeasible. Exercise to improve your estimation skills.
- If you need to generate all subsets of fixed size, look what the extreme cases are and use the "choose" function to determine the number of subsets you'd generate. If it's less than a few million, go for it. Btw, every good pocket calculator has the "choose" function built in. Don't use the bitfield counter method if N is too large, recursion or fixed for-loops are faster because you don't want to generate a lot of subsets that you don't use then because they have the wrong size.

Of course all these rules have their exceptions, for example if you need a long time to evaluate each built subset, then even for N=20 you might be too slow. Solve some problems to get a good intuition of what is feasible and what is not. Use the "Test" opportunity of the arena applet to test your solution with some worst case inputs. Look how long it runs (this is displayed in the test-result popup-window).

Summary

What have we done in this tutorial? We've seen what sets, subsets, k-subsets etc are and how to generate them recursively and iteratively with various methods. Methods to calculate the number of (k-)subsets were discussed. Then we fooled around with the methods a bit, both to achieve new things and to gain further insight into how it all works. I hope one thing you've seen as I saw it is that nothing is fixed, you can modify and adapt algorithms to satisfy your needs and sometimes to stumble across amazing discoveries. I had a lot of fun writing this tutorial and I sure as hell learned some new things, too. Thanks for listening, over and out.

Exercises

Yes, I actually want you to do exercises. They shouldn't be too hard, and they're well worth the time.

1. Solve the LightMeal problem in practice room 112 without looking at my solutions (but it's ok to look at the rest of the tutorial). Don't use optimizations yet. Do it in two different ways:

    a. Recursively without any global data, i.e. any external data the recursive function uses it must get as parameters.
    b. Recursively without any parameters, i.e. any external data the recursive function uses must be global. Not even the parameter "i" is allowed.

    For both versions, run the system tests to make sure you didn't make mistakes.

2. Which solution of the previous exercise do you like better? Why? Compare it to my solutions above, which are half global, half with parameters.

3. If you haven't done so already, use two optimizations for your LightMeal solutions: 1) Break if you already have enough vitamins and 2) break if the calories already exceed the best found answer so far. Run the systests again. Test if the optimizations help at all for an input of 20 times "5 5 5 5". Find inputs where they are extremely helpful. Try to find other optimizations. If you find one, tell me about it.

4. Solve some of the problems "Hiring" (125#2), "Big2" (125#3), "Diet" (58#2) (thanks JWizard) and "TugOfWar" (67#1). The numbers after the problem names tell you the practice room and problem numbers. For each problem, first explain what the problem asks for, using the set terminology we used here. Find out the maximum for N and other limits the problem specifies. Discuss which algorithm is appropriate. Run the system tests when you're done. Compare your solution to mine in my archive. Let me know if you find something I could be interested in.

5. I've illustrated generating the (k-)subsets in lexicographic order by showing you small example outputs. Make sure I didn't lie to you. Try larger examples and maybe *prove* that the algorithms work. Should you find a mistake, you can embarrass me, that alone should be worth it.

6. If you have any comments, suggestions or whatever about this tutorial, let me know.

Further Reading

If you know more resources, please tell me!

- "Combinatorial Algorithms : Generation, Enumeration, and Search" (pages 32-42) book by Donald L. Kreher, Douglas R. Stinson
- "Information on Subsets of a set" from "The (Combinatorial) Object Server"
- "The Algorithm Design Manual" (pages 117-118, 250-252) book by Steven S. Skiena (excerpt)
- Set, Subset, PowerSet, k-Subset, Choose and Multiset from MathWorld

---

Stefan Pochmann

Last modified: Mon Jun 10 04:12:38 PDT 2002