The Four Basic Algorithms and Backtracking

Nadeem Mohsin

September 14, 2013

1 Contents

This handout covers the four basic algorithms we traditionally always teach at our first lecture. Of these four, one is mathematical (Euclid's algorithm for the greatest common divisor), while the other two are brute force combinatorial algorithms for finding subsets, permutations, and combinations. We'll deal with the GCD algorithm first, and then move on to the three brute force techniques, which are actually just three examples of the general technique of backtracking.

2 Finding the greatest common divisor

Keep in mind that we'll be dealing exclusively with integers here, and non-negative integers in most cases. The few times negative integers come up in such problems, it's generally pretty obvious what to do with them.

We'll use the notation $p \mid a$ to indicate that the integer p cleanly divides the integer a. For example, $3 \mid 15$.

A *common divisor of* a *and* b is any integer p such that $p \mid a$ and $p \mid b$. As an example, 2 is a common divisor of 12 and 18. 3 is another such common divisor. And 1 is always a common divisor of any pair of numbers. In general there are many common divisors, but the one with the most interesting properties is the *largest* one - the *greatest common divisor* or GCD for short. In the previous example, $\gcd(12, 18) = 6$.

How might we go about computing the GCD of two numbers a and b? It's pretty easy to think of a few - if the numbers aren't too large, just iterate from 1 to $\min(a, b)$ and try every number. The biggest one that divides both is the one we want. Another way might be to prime-factorize both numbers and figure it out from their factors.

Still, most of these approaches are relatively slow in comparison to Euclid's algorithm. This is probably the oldest algorithm you will ever learn - it was first written down 2300 years ago, and was probably known before then. As a plus, it's literally a one-line recursive function to implement.

The algorithm is based on the following relation. Let r be the remainder when a is divided by b - i.e, $r = a \bmod b$.

$$\gcd(a, b) = \begin{cases} \gcd(b, r), & \text{if } b > 0 \\ a, & \text{if } b = 0 \end{cases}$$

And that's the entirety of the algorithm! Simply implement this as a recursive function and you're done.

```
int gcd(int a, int b) {
  return b == 0 ? a : gcd(b, a % b);
}
```

The full analysis of its runtime is a bit complicated, but a decent upper bound is logarithmic in the input values. So even with longs, the algorithm is practically instantaneous.

In case you're curious about why this works, read on. Otherwise, feel free to skip the next bit.

2.1 Explanation

Think back to the basic definition of division that you used in school, when they threw around terms like 'quotient' and 'remainder'. Say we divide a by b, and the quotient is q, while the remainder is r. This can be expressed compactly like this:

$$a = bq + r, \text{ where } 0 \leq r < b$$

Simple example: Let $a = 32$ and $b = 5$. Then the quotient is 6 (32/5 in code) while the remainder is 2 (32%5 in code).

$$32 = 5 \cdot 6 + 2$$

We don't know what $\gcd(a, b)$ is just yet, but let's just call it g. (A surprising amount of math and computer science just comes from pretending to know things you don't.) But what does this mean? It means that a is a multiple of g, and b is also a multiple of g.

Since we can express $r = a - bq$, this means that r is the difference of two multiples of g. It should be pretty obvious that this means it too must be a multiple of g.

This is where things get interesting. Since g is a divisor of both b and r, it's a (wait for it) common divisor! At this point you should immediately get suspicious, and wonder if it might be the greatest common divisor of b and r.

Your wishful thinking will pay off, because this is in fact the case. How can you be sure? Simply fall back on the computer scientist's favorite tool - *proof by contradiction*. Since we're ornery types by nature, we'll just assume that g *isn't* the largest common divisor. So there must be some other common divisor h such that $h > g$. What is h? I don't know, but like I said earlier, we love to pretend to know things we don't, and far be it from me to break with tradition.

Now see what happens. Since $a = bq + r$, we can see that a must be a multiple of h too - since it is the sum of two multiples of h. But if that were the case, then how can g be $\gcd(a, b)$? We have here a number h that is a common divisor of both, and bigger than g.

That's a contradiction, which means our initial assumption was flawed. No such number h can exist. This is what happens when you pretend to know stuff, you see. Anyway, getting back to our original argument, this means that $g = \gcd(b, r)$.

What we've done here is replace our original problem ($\gcd(a, b)$) with something a little smaller ($\gcd(b, r)$). We can always repeat this process - the new a is the old b, and the new b is the old r. Rinse and repeat.

We keep going until one of the numbers eventually hits zero. When that happens, we just use the fact that $\gcd(a, 0) = a$, and that's the base case for our recursion. Woohoo, proof complete.

2.2 Finding the least common multiple

A problem very closely related to GCD is LCM - the least common multiple of two integers. Just as with common divisors, we can define common multiples. The common multiples of a and b are divisible by both of them. There's obviously an infinite number of these - for example, some common multiples of 6 and 10

are 30, 60, 90, and so on, forever. We care about the one that's as small as possible - the *least* common multiple. So $\mathrm{lcm}(6, 10)$ would be 30.

To find the LCM, just find the GCD, and then use this neat fact.

$$\gcd(a, b) \times \mathrm{lcm}(a, b) = a \times b$$

Since we've already got Euclid's algorithm up our sleeve, LCM is now a pretty trivial problem too.

3 The fine art of combinatorial brute force

For this section, we're going to be building some things out of a list L that has N elements. It doesn't really matter what's in L - the most we care about is the ability to index into L as if it were an array.

Given this list, we would like to find all possible subsets of L. Or all possible permutations, or all possible combinations. The specific details in each case will look a bit different, but the underlying technique is the same - a simple example of what we call *backtracking*.

The way backtracking works is that we build up a solution piece-by-piece. Sometimes we realize that our current solution isn't going to get us what we want, in which case we 'backtrack' by going back a few pieces.

The key idea is that we're making a *sequence of choices*. Making all the choices gives us a complete solution. Partial solutions are just an incomplete sequence of choices. And when we backtrack, we're just figuring out that some of the last few choices were mistaken. So we just undo the last few decisions until we're happy, and continue building our solution by making different choices this time. All this usually happens very neatly with the magic of recursion, so don't worry about horrendously complicated implementations.

3.1 Basic terminology and examples

A *subset* of L is just a bunch of elements of L. We don't care about their order, as long as you don't reuse them. You don't have to use all of the elements.

The subsets of $\{a, b, c\}$ are $\{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}$, and the empty set $\{\}$. So 8 subsets in total. This happens to be 2^3, which is somewhat suggestive.

In general, a list of N elements will have 2^N subsets. This is an exponential function, so if you're generating all possible subsets, you really can't go too far above $N = 20$ or so. It's rare to even see $N = 25$. Sometimes you can get away with a few clever hacks and optimizations, of course. But don't go around trying to generate all subsets on a list of 40 elements - you'll just time out, and I'll have to commit *seppuku* in shame.

A *permutation* of L is just any reordering of the elements of L. You have to use all the elements, of course.

The permutations of $\{a, b, c\}$ are (with some abuse of notation) $abc, acb, bac, bca, cab, cba$. That's $6 = 3!$ permutations.

A list of N elements has N! permutations. This grows even faster than the exponential function, so generating all permutations is not really feasible beyond $N = 10$ or so. You can probably go up to 11 or 12, but beyond that you're pretty much asking for trouble with the time limit.

A *combination* of L is basically a subset of some fixed size. More precisely, a combination of size k is any subset of L with exactly k elements.

The combinations of size 2 of $\{a, b, c\}$ are $\{a, b\}, \{b, c\}, \{a, c\}$. Notice that there are $3 = \binom{3}{2}$ elements.

A list of N elements has $\binom{N}{K}$ combinations of size K. It may be helpful to know that for a fixed N, the highest value of $\binom{N}{K}$ is at $K = N/2$. Often this is not as bad as pure subsets, so as long as you don't generate all subsets, you can often do combinations for slightly larger values of N. If the max value of K is relatively small, that helps a lot.

For example, generating all subsets of a list of 30 elements is out of the question, but a combination of size 8? Pretty doable - there are only about 6 million of those.

3.2 The algorithms

Each of these problems reduces to making a different sequence of choices. Keep that in mind and the implementation will follow from that.

3.2.1 Generating all subsets

To generate any subset, we basically walk along the list, and for each element, we have to make a binary choice - is it in our subset, or not? Each of these N choices can be either yes or no, and every configuration of N such decisions will give a different subset.

For example, if $L = \{a, b, c, d\}$, then the subset $\{a, c, d\}$ is the result of the choices "yes, no, yes, yes", indicating that we take everything except b. On the other hand, the sequence "no, no, yes, yes" would give the subset $\{c, d\}$.

Having recognized this, we can now create a backtracking solution. Remember, this is a *sequence* of choices - we make them in order, one after the other. Bearing that in mind, here is a little code snippet that generates all possible subsets of L.

```
void subsets(int p, boolean[] in) {
  if (p == N) {
    // The array 'in' represents our subset.
    // Do something problem-specific here, maybe print it.
    return;
  }
  // L[p] is not in the subset.
  in[p] = false;
  subsets(p + 1, in);
  // L[p] is in the subset.
  in[p] = true;
  subsets(p + 1, in);
}
```

Here the boolean array tells us which elements are in the set and which are not. The variable p is just an index into the list as we walk from left to right. Our recursion bottoms out when we've made all N choices. At this point we have a subset, and can do whatever - print it, evaluate it in some way, do something else.

At the pth element, we make both possible choices for L[p] and recurse on the remainder of the problem. The first recursive call will generate all the subsets that don't include L[p]. The second will generate all the ones that do include it.

This isn't the only way we might want to do it. For example, we could just build up the subset in an array instead of storing it as a boolean array.

```
void subsets(int pos, int len, int[] sub) {
  if(pos == N) {
    // Do something problem-specific with the subset here.
    return;
  }
  // Try the current element in the subset.
  sub[len] = L[pos];
  subsets(pos+1, len+1, sub);

  // Skip the current element.
  subsets(pos+1, len, sub);
}
```

This assumes L is an array of integers, but you can modify things pretty easily from here.

There's a slightly different way to code up the first approach that is of great interest. Instead of using a boolean array, we'll just use a single integer whose individual bits represent the entries of the array. So the ith bit is 1 or 0 as the ith entry of the array is true or false. This is usually called a *bitmask*, and will show up over and over again in other types of problems. Bitmasks are really handy, so get comfortable with them.

```
void subsets(int pos, int mask) {
  if(pos == N) {
    // Do something problem-specific with the subset.
    // For example, you can print it.
    for(int k = 0; k < N; k++) {
      if(on(mask, k)) {
        System.out.printf("%d ", L[k]);
      }
    }
    System.out.println();
    return;
  }

  // Skip this element.
  subsets(pos+1, mask);
  // Try putting it in the subset.
  subsets(pos+1, set(mask, pos));

}

// Check if the bit at index pos is on in mask.
boolean on(int mask, int pos) { return (mask & (1 << pos)) > 0; }

// Set the bit at index pos in mask.
int set(int mask, int pos) { return mask | (1 << pos); }
```

If you're observant, you'll have noticed a truly epic shortcut that can eliminate the need for backtracking altogether. It's too useful not to share, so we'll cover it here. The key observation is that every string of N

bits represents a different subset of L. All our code is doing is generating all N-bit integers.

But the N-bit integers are just the numbers from 0 (all N bits zero) to $2^N - 1$ (all N bits one). So the entirety of our code reduces to a single loop.

```
void subsets() {
  // mask will iterate through all 2^N subsets.
  for(int mask = 0; mask < (1 << N); mask++) {
    // Do something problem-specific with the subset. Here I'll just
    // print it.
    for(int k = 0; k < N; k++) {
      if(on(mask, k)) {
        System.out.printf("%d ", L[k]);
      }
    System.out.println();
  }
}
```

As they say on the internet (my native land), bit magic is best magic. Won't be the last example of bit-twiddling you'll see in this business.

3.2.2 Generating all combinations

For generating combinations, our sequence of choices is mostly the same as with subsets. The only difference is that now we won't continue if we realize that it's impossible to build a combination from our partial solution. This is often called *pruning*. The reason for this name is that we can look at the recursive calls as a big tree, in which we're just cutting out whole branches that won't go anywhere useful. Pruning is a common optimization when doing more complicated backtracking problems.

As before, we will move through the elements of L in sequence. This time we have to limit the size of our subset to K, so we'll also keep track of the current size of our partial combination. If we notice that there aren't enough elements remaining in L to complete our combination, we don't need to continue down that line of exploration any longer.

```
void combine(int p, int size, int[] comb) {
  // Prune away useless branches.
  if (N - p < K - size) return;

  if (size == K) {
    // Do the problem-specific stuff here.
    return;
  }

  // Skip L[p].
  combine(p + 1, size, comb);
  // Use L[p] in the combination.
  comb[size] = L[p];
  combine(p + 1, size + 1, comb);
}
```

As an exercise, you can try to modify this code to use bitmasks instead. Another thing you might do is to get it to print out combinations in lexicographic order. It's quite common to see a problem asking for the lexicographically smallest solution. In other cases, any solution will do.

3.2.3 Generating all permutations

In this problem, the sequence of choices we'll make is somewhat different from the last two algorithms. Imagine that we're building a permutation one element at a time. At each step, we will pick an unused element for the current position and move on to the next position.

```
void permute(int p, boolean[] used, int[] perm) {
  if (p == N) {
    // Do something problem-specific with perm here.
    return;
  }
  // Try every unused element in the current position.
  for (int i = 0; i < N; i++) {
    if (!used[i]) {
      used[i] = true;
      perm[p] = i;
      permute(p + 1, used, perm);
      used[i] = false;
    }
  }
}
```

Don't let the loop throw you off. The idea is the same as in the other algorithms - try to make the choice in all possible ways. It just so happens that there are more than two choices here - the first position has N choices, the second has $(N - 1)$ and so on. So we have to run through all the possibilities with a loop and recurse appropriately. At the bottom of our recursion tree, the perm array will represent a complete permutation.

It's fairly straightforward to replace the boolean array with a bitmask if you want to.

It is possible to do permutations without backtracking as well. Here is an iterative algorithm that takes any permutation and generates the lexicographically next permutation. Just start with the original list and call it repeatedly. The implementation below is smart enough to recognize when it has reached the lexicographically last permutation too. (Hence the boolean return type.)

I'll leave this to the reader to figure out. But an easier thing to do is generate this algorithm yourself. Just make a lexicographically sorted list of the permutations of 1, 2, 3, 4, say, and figure out the pattern that controls how they change. It's an excellent exercise, and as a plus, you'll be able to come up with this again even if you don't have the code at hand.

```
boolean next_permutation(int[] nums) {
  int n = nums.length;

  int i = n-2;
  while(i >= 0 && nums[i] >= nums[i+1]) i--;

  if(i == -1) return false;

  int x = nums[i];
  int j = n-1;
  while(j > i && nums[j] <= x) j--;

  nums[i] = nums[j];
  nums[j] = x;
```

```
    Arrays.sort(nums, i+1, n);
    return true;
  }
```