Recursion

William E. Skeith III

Abstract

In this lecture we describe an altogether new technique for writing programs called *recursion*. Simply put, a **recursive function** is one which makes *calls to itself*. While the technique is never strictly necessary (any program which could be written recursively could also be written without recursion), it is nevertheless an indispensable tool. As we will see, some problems have natural recursive solutions that would be rather awkward to implement iteratively.

Section 1

Proofs by Induction

Before diving into recursive programs, we review the closely related topic of mathematical induction. Recall the typical situation for an inductive argument: you usually want to prove some *parameterized* statement $T(n)$ for all natural numbers $n \in \mathbb{N}$. For concreteness, here are a few examples of such statements. $T(n)$ could be the statement that

- A sum or product has a particular closed form, e.g, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- A property of a set of size n, e.g., $T(n)$ might state that the number of subsets of any set of size n is 2^n.
- The existence of some object of size (related to) n, e.g., $T(n)$ could assert the existence of a $2^n \times 2^n$ matrix H with all entries equal to ± 1 and all rows of H orthogonal.
- The correctness of a program: e.g., program P on any input x of size n will produce the desired output $f(x)$ (where f is some mathematical description of the desired functionality).

Before going further, we review the high level blueprint for inductive proofs. Let's say we want to prove that for all $n \in \mathbb{N}$, the statement $T(n)$ is true. A typical inductive proof would proceed with the following steps:

1. Prove explicitly that $T(1)$ is true.
2. *Assuming* that $T(m)$ is true, prove that $T(m+1)$ would also be true.
3. By iteration of the above steps, it follows that $T(n)$ is true for all $n \in \mathbb{N}$.

This is not a "traditional" proof like you might have seen in middle school trigonometry class (where in particular the number of steps is fixed). However, this argument *proves that such a proof exists*. Perhaps think of it as a *proof generator*: for any particular value of n, this machinery shows that you could construct the following "traditional" proof (the numbers underneath reference items (1-3) above):

$$\underset{(1)}{T(1)} \underset{(2)}{\implies} T(2) \underset{(2)}{\implies} T(3) \underset{(2)}{\implies} \ldots \underset{(2)}{\implies} T(n-1) \underset{(2)}{\implies} \underset{(3)}{T(n)}. \tag{1}$$

If you're in the mood for lame analogies, try this: imagine there is some delicious cheese up high on a platform[1] (let's call this platform n). It looks treacherous, and you don't know how you would get up there, but you really want the cheese. You notice that there are a bunch of other platforms, and adjacent platforms aren't that far apart. In fact, with a small plank, you could build a bridge between any two adjacent platforms. Moreover, one of these platforms actually looks pretty easy to climb. So you climb up there with your plank and repeatedly bridge the gaps until you arrive at n and receive your reward.

A few remarks are in order. First it should be noted that inductive proofs do not have to proceed in such a linear fashion. In the analogy, we only used each bridge once, but that won't always be the case. (We will see some examples soon.)

Second, note that the inductive proof merely shows that obtaining the cheese is *possible*: you prove that you have an appropriate plank and that you could get yourself to the first platform, and so *in principle*, if you were sufficiently motivated, you could indeed get to platform n. However, in light of the non-linearity we just alluded to, it may require many steps (many more than n in some cases) to complete the task. And herein lies one of the really fun things about writing recursive programs vs inductive proofs: the computer will actually *run your proof* and go fetch your cheese! Again, plenty of examples are on the way, but first some exercises.

Exercises

1. Prove by induction that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all $n \in \mathbb{N}$.
2. Prove by induction that for any number r, $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$ for all $n \in \mathbb{N}$.
3. Prove by induction that the number of subsets of a set of size n is precisely 2^n. (Note that the empty set counts as a subset.)
4. Prove by induction that for all $n \geq 0$, there exists a $2^n \times 2^n$ matrix with all entries in $\{-1, 1\}$, and all rows orthogonal (meaning the inner product of any two **distinct** rows is 0). *Hint:* if you manage to find one such $m \times m$ matrix H, show how you could form a related $2m \times 2m$ matrix which also has this property. If you need even more of a hint, it might look something like this, for some related H': $\left(\begin{smallmatrix} H & H \\ H & H' \end{smallmatrix}\right)$.

[1] Some imagination may be required to concoct a plausible story explaining how the cheese got there and how you managed to know about it.

Section 2

Intuition

Imagine you are solving a problem that satisfies the following criteria:

1. Each input has an integer *size* associated to it. For example this could be the size of a vector, the length of a string, or perhaps the input itself (if it is an integer).
2. For tiny values of the size, the problem is easy to solve.
3. If given a *magic black box* that solves the problem for smaller inputs (of size $< n$), you would be able to solve inputs of size n. (The problem has some kind of self-similarity.)

This enables the following approach to programming a solution for inputs of size n, provided you can write functions that call themselves:

1. "Hard-code" the solution for a minimal value of the size.
2. If the input isn't tiny enough to be covered by step 1, then use *your own function* as the magic black box to solve one or more **smaller** instances, and assemble those solutions to form your solution.
3. Observe that since the recursive calls always have **strictly smaller inputs**, any sequence of calls must eventually terminate at step 1. Hence, if your reassembly in step 2 is correct, the whole thing works!

Step 2 is a bit strange, as you are calling this function that you haven't even finished writing, and yet you assume it will magically work! But if you take a step back and look at the big picture, you'll see there is no contradiction in all this, as pointed out in step 3. It is very much like a proof by mathematical induction. Note that steps 1-3 are almost identical to the steps that comprise an inductive proof. In fact, you can almost look at your function as an *inductive proof of its own correctness*. Let's explore some examples in detail.

Section 3

Examples

3.1 Warm ups

To begin, let's write a recursive function to compute $n!$ (the factorial function). It is trivial to do this without recursion, but it provides an easy first example to analyze. Here's the key observation: if I gave you a magic black box that computed $m!$ for any smaller number $m < n$, how could you use it to help you compute $n!$? Well, note that $n! = 1{\cdot}2{\cdot}3{\cdot}\ldots{\cdot}n = (n-1)!{\cdot}n$. So, you could use the magic black box to compute $(n-1)!$ and then just multiply by n. Perhaps something like this would work:

```
unsigned int fac(unsigned int n)
{
    return fac(n-1)*n;
}
```

This is of course ridiculous, as it would require actual magic, but it is very close to something that *does* work! So close, it might still seem like magic. Here is a working solution:

```
unsigned int fac(unsigned int n)
{
    if (n == 0) return 1; /* Note that 0! = 1 */
    return fac(n-1)*n;
}
```

The only difference is that we explicitly defined what happens for one input (0). For all other inputs, recursive calls have to be made. Notice how much work the computer does for us: we simply give it the "bridge" which computes $n!$ given $(n-1)!$, and a single starting point, and it will figure out a path from the starting point to the value we want.

It is crucial that you have a solid mental model of what is happening when you call this function. Say we attempt to evaluate `fac(3)`. This was defined in terms of `fac(2)`, which was defined in terms of `fac(1)`, which was defined in terms of `fac(0)`. Indeed, at some point in time, there will be **four** copies of our function running with different inputs.[2] `fac(0)` makes no additional calls, and returns the value 1 to `fac(1)`, which sets its return value to `fac(0)*1 = 1*1 = 1`. This is then given to `fac(2)` which computes its return value as `fac(1)*2 = 1*2 = 2`, and finally `fac(3)` can compute our desired result as `fac(2)*3 = 2*3 = 6`.

The following simple program might help you understand. See if you can guess the output of `f(3)` where `f` is defined as follows:

```
void f(int n)
{
    printf("calling f(%i)\n",n);
    if (n == 0) return;
    f(n-1);
    printf("leaving f(%i)\n",n);
}
```

Ready to see how it plays out? To save you a trip to your terminal, here is what would happen:

```
calling f(3)
calling f(2)
calling f(1)
calling f(0)
leaving f(1)
leaving f(2)
leaving f(3)
```

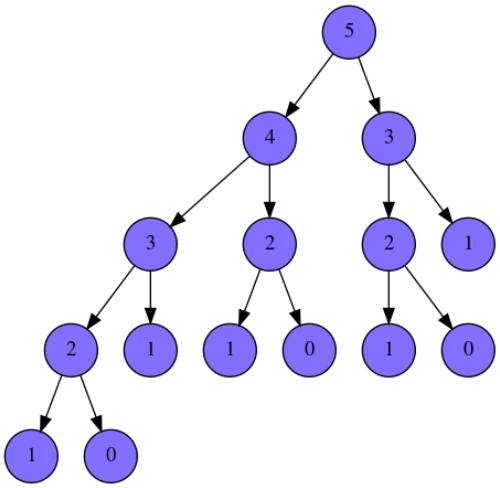
[2]Note that all but one copy of the function will be "frozen" while waiting for one of the others to return a value to it.

Notice that the "copy" of `f` with input 3 cannot finish execution until `f(2)` is computed, and so on. Hence the nesting effect. An important lesson can already be learned from this: there is a hidden cost in terms of *memory* that you must consider when writing recursive functions. In particular, the "frozen state" of all the functions that are on hold must be kept somewhere.[3] Hence, your program's memory consumption will be at least proportional to the longest chain of calls it makes.[4]

Let's now take a look at a *non-linear* situation, where more than one recursive call is made in the body of the function. Recall the *Fibonacci sequence* (1,1,2,3,5,8,13,21...) that we studied in lecture 2. There is of course a very natural way to implement this recursively, as its very definition was recursive. The following will indeed compute the terms of the sequence correctly:

```
int fib(int n)
{
    if (n < 2) return 1;
    return fib(n-1) + fib(n-2);
}
```

Wow! So elegant! But we might want to have a closer look at this before we get too carried away patting ourselves on the back. If you run this on small inputs, everything looks peachy. But for inputs around 50, you might notice that the answer is not immediately produced, and for some reason your laptop is feeling rather warm... Let's investigate by **tracing** our program. In particular, let's draw the **recursion tree**, which is a graph of all the function calls that were made during execution. If you think carefully about this, you will see that to figure out `fib(5)`, the following calls had to be made:



[3] This "somewhere" is known as the *call stack*, which we will discuss more in class.

[4] There are some exceptions to this: most compilers implement *tail recursion*, which converts recursive calls into a loop when it is convenient. In particular, this conversion is triggered if the compiler can guarantee the body of your function will make at most one recursive call, which moreover happens at the end (upon the function's return). For example, if our factorial function were compiled with optimization enabled, it would be reduced to a loop in the assembly code and require only a constant amount of memory.

Yikes! You can see the issue: the function has no memory of computing `fib(m)`, and so it computes this value again and again *from scratch*. Each time the input n increases by 1, the number of calls required will approximately double, leading to an exponential number of calls in total. Hence the warmth of your laptop. There is a straightforward, general technique—called *memoization*—for converting inefficient recursive functions like this (where the recursion tree has a lot of duplication) into efficient ones. As the name suggests, it involves supplying your function with more memory, but for now we'll leave the details to your imagination.

3.2 Merge Sort

I think it's about time we take a look at a situation where recursion actually is a good idea and produces a solution that is both elegant and efficient. Let's revisit the problem of sorting an array. Back in lecture 3 we used the *selection sort* algorithm for this task. It was conceptually simple, but it actually required far more steps than necessary (on an array of size n, it required roughly n^2 operations). Here's a new idea: divide the array in half, and via some recursive magic, sort the two halves independently. Lastly, reassemble the two sorted halves into one sorted array.

Let's look at the reassembly step first. Imagine you have two stacks of cards, and each one is sorted. Putting them into one sorted stack is relatively easy: start a third stack for the result, and then just examine the top card from each stack to decide which one goes next into the third.[5] This is a good review exercise. You should try it out on your own before reading ahead. When you're ready, here is one way to do it:

```
/* A and B are sorted (ascending) arrays of size nA, nB respectively.
 * We will place the result into S, which we assume to have sufficient
 * space (at least nA+nB elements). */
void merge(int* A, int* B, size_t nA, size_t nB, int* S)
{
    size_t iA = 0; /* index of smallest unplaced item from A */
    size_t iB = 0; /* index of smallest unplaced item from B */
    size_t iS = 0; /* location to add next element into S */
    while (iA < nA && iB < nB) { /* both stacks have unplaced elements */
        if (A[iA] < B[iB]) S[iS++] = A[iA++];
        else S[iS++] = B[iB++];
    }
    /* either A or B has run out of elements.  add whatever is left: */
    while (iA < nA) S[iS++] = A[iA++];
    while (iB < nB) S[iS++] = B[iB++];
}
```

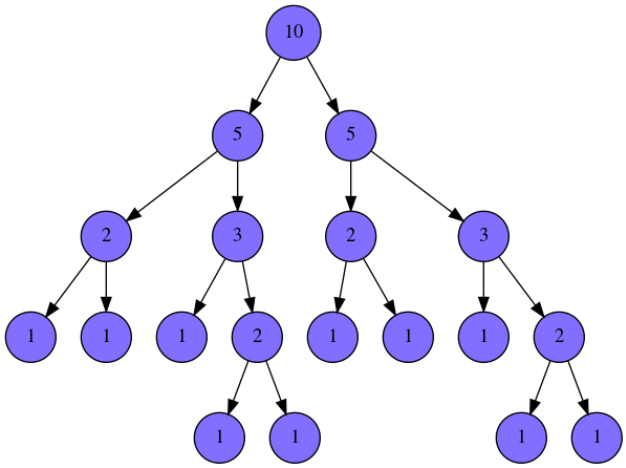
Thanks to the magic of recursion, we're quite nearly done already! Let's see how to put the final pieces into place. Remember that we need to handle small inputs explicitly (by the

[5] You can place cards into the third stack face-down to preserve the order.

way, this is often called the **base case**). For arrays of size 0 or 1, they are already sorted, so there's nothing to do! This gives us the following solution:

```
/* NOTE: aux should point to an array as large as A.  we will
 * use aux to perform the merge... */
void mergeSort(int* A, size_t n, int* aux)
{
    if (n < 2) return; /* already sorted; nothing to do. */
    /* time for recursive magic: use our own function to
     * sort smaller sub-arrays: */
    mergeSort(A,n/2,aux);       /* sort left sub-array */
    mergeSort(A+n/2,n-n/2,aux); /* sort right sub-array */
    /* merge results back into aux: */
    merge(A,A+n/2,n/2,n-n/2,aux);
    /* finally, copy from aux back into A: */
    for (size_t i = 0; i < n; i++)
        A[i] = aux[i];
}
```

And here is the recursion tree when sorting an array of 10 elements (the number in each node shows the size of the array being sorted):



Lastly, let's consider the cost. It is relatively easy to see that $\approx n$ steps are required to merge two sorted arrays of length $n/2$ into one sorted array of length n: each iteration of the main loop requires only a constant number of operations, and places one element into its proper location in the sorted destination array. We will do merge quite a few times, but most of the calls to merge are on tiny arrays. For convenience, let's assume that the size of the array is a power of two, say $n = 2^k$. Then we will make one call to merge which costs n (the final merge), 2 calls which cost $n/2$ (just before the final merge), 4 calls of cost $n/4$, and so on. Note that after k iterations, we will hit the base case, as the subarrays in question

will be of size $n/2^k = 1$. So, we are left with the following approximation of the cost:

$$C(n) \approx \sum_{i=1}^{k} 2^i (n/2^i) = kn = n \log n.$$

For large values of n, this is far superior to selection sort! Furthermore, it can be shown (you'll see this result later on in your algorithms class) that this is in some sense as good as you can hope for: there is no *generic* algorithm[6] that sorts numbers substantially faster.

3.3 Greatest Common Divisors

Let's look at the slightly quirky example of computing greatest common divisors. For $a, b \in \mathbb{Z}$, we define the *greatest common divisor* of a, b as the largest integer d such that $d \mid a$ and $d \mid b$. (**Remarks on notation:** the notation $d \mid a$ means that $\exists m \in \mathbb{Z}$ such that $a = dm$. Also, we will denote the greatest common divisor of a, b by $\gcd(a, b)$, or more simply by (a, b).) An equivalent definition of $d = \gcd(a, b)$ is that it is the unique positive integer with the property that for all integers d' such that $d' \mid a$ and $d' \mid b$, then $d' \mid d$. Yet another way to define the gcd is as follows:

$$\gcd(a, b) = \min_{\substack{x,y \in \mathbb{Z} \\ xa+yb>0}} \{xa + yb\}. \tag{2}$$

That is, it is the smallest nonzero *integer* linear combination of a and b.[7] (See exercise 9 for a proof of the equivalence of these definitions.) Next recall the so-called *division algorithm*, which is simply the fact that $\forall a, b \in \mathbb{Z}$ with $b \neq 0$, $\exists q, r \in \mathbb{Z}$ with $0 \leq r < |b|$ such that $a = qb + r$. Moreover, q, r are unique. (Note: q stands for **q**uotient, r for **r**emainder.)

We wish to find an efficient algorithm for computing gcd's, even for extremely large integers (say, hundreds of digits!). We will make use of the following observation, with q, r as above: the common divisors of a and b are the same as the common divisors of b and r. That is, for any d such that $d \mid b$, we have

$$d \mid a \iff d \mid r. \tag{3}$$

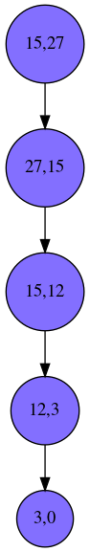
(**Exercise:** prove this.) This observation tells us that to compute (a, b), we could just as well compute (b, r), as long as $b \neq 0$ (so that r is defined). This seems potentially valuable since $r \lneq b$, and indeed it is: this observation essentially **is** the algorithm. Here's how we can reason out the remaining details: we'll define the *size* of an input a, b as the size of the second parameter b. If we use $\gcd(b, r)$ in order to compute $\gcd(a, b)$, notice that the second parameter is guaranteed to be smaller. And what if $b = 0$? This is our base case! Since everything divides 0, $\gcd(a, 0) = |a|$. To simplify things a bit, let's assume that all inputs will be non-negative (that is, $a, b \geq 0$). Our discussion above suggests the following elegant (and efficient!) procedure:

[6] That is, one that does not make some additional assumptions about the input data.

[7] This explains the notation (a, b): in general (a, b) stands for all integer linear combinations of a, b and similarly (d) stands for all multiples of d. If d is the greatest common divisor of a, b, then $(a, b) = (d)$.

```
unsigned int gcd(unsigned int a, unsigned int b)
{
    if (b == 0) return a;
    return gcd(b,a%b);
}
```

The following shows a trace of $\gcd(15, 27)$. Note that if $a < b$, the first recursive call has the effect of swapping the parameters.



While knowing $d = \gcd(a, b)$ already seems useful, for some applications (like computing modular inverses, which is a common task in cryptography) it is essential to know *how* d is the greatest common divisor. In particular, we want to know $x, y \in \mathbb{Z}$ such that $xa + yb = d$. With a little work, we can modify the above algorithm to compute suitable[8] integers x, y. If you think about it, the function we want to write has *multiple outputs*. At a minimum, it should give us x, y but let's also output $d = (a, b) = xa + yb$. It might not be obvious how to return multiple values in a clean way using C/C++. There are several options, but one common idiom is as follows: often when a function has multiple outputs, the mechanism to deliver them is via pointers, or via reference parameters. Here's a possible prototype for what we want:

```
unsigned int xgcd(int& x, int& y, unsigned int a, unsigned int b);
```

Then a call to this function would look something like:

```
int x,y; /* storage for additional values "returned" by xgcd */
unsigned int a = 13, b = 23;
unsigned int d = xgcd(x,y,a,b);
/* NOTE: at this point x and y are set so that xa + yb = gcd(a,b) */
printf("gcd(%i,%i) = %i = %i*%i + %i*%i\n",a,b,d,x,a,y,b);
```

[8]Note that x, y are not unique. If given one pair, can you describe the others?

With that technical detail out of the way, the big question is how do we compute x, y? As usual, imagine you have a magic black box that does what you want for any inputs smaller than the one you are given. Recall that in this case "smaller" means b is smaller, as this is how we defined the size of an input. A little thought is required, but the solution is actually fairly straightforward. Assuming our magic box works, if we were to call `xgcd` on b, r, we could obtain integers x', y' such that

$$x'b + y'r = d$$

where $d = \gcd(b, r) = \gcd(a, b)$. That's nice, but we need an integer linear combination of a, b, not of b, r. However, using the division algorithm we have $a = qb + r$, and hence we can express r in terms of a, b: $r = a - qb$. Now we're getting somewhere! Observe that

$$\begin{aligned} d &= x'b + y'r \\ &= x'b + y'(a - qb) \\ &= x'b + y'a - y'qb \\ &= y'a + (x' - y'q)b. \end{aligned}$$

So, y' is the new x, and $(x' - y'q)$ is the new y! Here it is in program code:

```
unsigned int xgcd(int& x, int& y, unsigned int a, unsigned int b)
{
    if (b == 0) { /* base case: gcd(a,b) = a = 1a + 0b */
        x = 1;
        y = 0;
        return a;
    }
    int xx,yy; /* x' and y' */
    unsigned int q = a/b, r = a%b;
    unsigned int d = xgcd(xx,yy,b,r);
    /* xx and yy are the right values for b and r, but we need values
     * for a and b.  Since a = qb + r, we have the following: */
    x = yy;
    y = xx - yy*q;
    return d;
}
```

Are you feeling the magic of recursion yet? I know I am. Let's prolong the magic with a few more instructive examples.

3.4 Combinatorics

Consider the problem of computing all *permutations* of a string. For example, if the input is the string "abc", then we would like to produce all rearrangements as output:

abc
acb
bac
bca
cab
cba.

Once again, imagine you have a magic black box that works on any smaller string (we'll use the length of the string as the size of an instance). If you have a string s of n elements, how could you use the magic box on strings of length $n-1$ to help you figure out all permutations of your string s? The ordering of the example output above is actually a hint. Look at the first two pairs. Notice the first element stays put, and the last two swap. Put another way, the last two elements are placed *in every possible arrangement*, which is something we could use a recursive call to figure out! Our approach will be to give every element a "turn" going first, and use the magic box to run through all possible arrangements of the remaining $n-1$ elements. The programming details are a little gross in this case, but the idea is actually quite simple. Here's an outline. Start with an empty list of strings P to store the output. Then for i in the range $0, \ldots, n-1$ perform these steps:

1. Swap $s[0] \leftrightarrow s[i]$.
2. Compute all permutations of the suffix $s[1, \ldots, n-1]$.
3. Add to output P each string produced in step 2 but with $s[0]$ prepended.

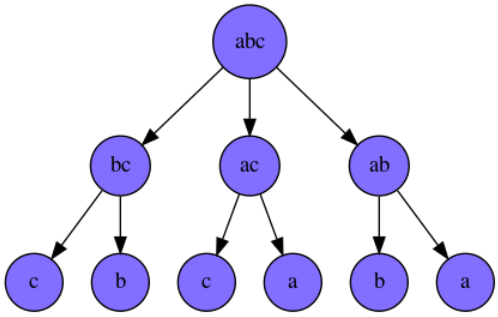
The only issue of course is that we never took care of a base case. But for short strings, this is easy: if the length is less than 2, there is only one permutation, which is the string itself. (*Note:* on the empty string, the correct answer is `[""]`, not `[]` if you think about it.) A prototype might be as follows:

```
vector<string> perms(string s);
```

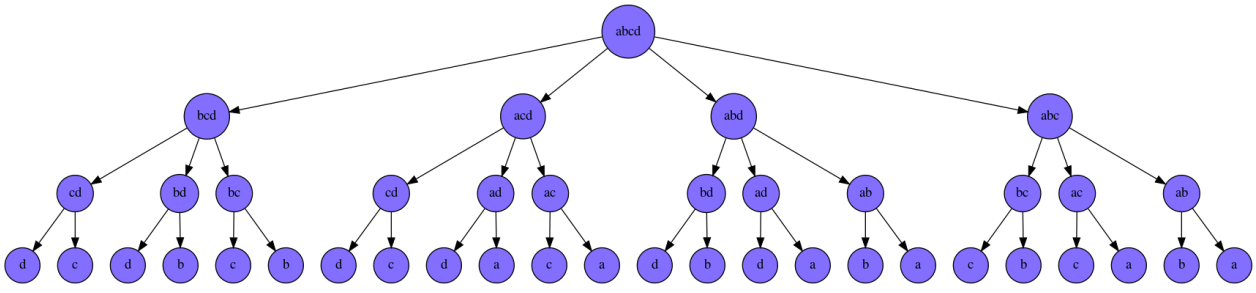
This is an excellent exercise. The reader is encouraged to work out the details right now before looking at the solution below. You will probably find the `substr` function of utility: `s.substr(1)` will return a string containing the elements `s[1,..,n-1]` (the first element is omitted). Below is one possible C++ implementation of this idea.

```
vector<string> perms(string s)
{
    vector<string> P; /* return value */
    if (s.length() < 2) { /* base case; return [s] */
        P.push_back(s);
        return P;
    }
    for (size_t i = 0; i < s.length(); i++) {
        swap(s[0],s[i]); /* i's "turn" to go first */
        /* now use recursive magic to compute permutations of suffix: */
        vector<string> T = perms(s.substr(1));
        /* assuming the above worked, T has all permutations of the
         * of s[1..n-1].  Add s[0] to the front of each and store in P: */
        for (size_t j = 0; j < T.size(); j++)
            P.push_back(s[0] + T[j]);
    }
    return P;
}
```

Here is the recursion tree if you were to call this on the string "abc".



And here is the recursion tree if you were to call this on the string "abcd"!



Notice how the number of recursive calls made depends on the length of the input (none of our prior examples had this feature, beyond the fact that the base case made no recursive calls). Note also that each of the $n!$ leaf nodes determines a unique permutation: label the

arrows by which element has been chosen to be "first", and then the path from root to leaf spells out a permutation (adding an implicit arrow out of each leaf[9]).

Exercises

1. Write a recursive function to compute a^b for non-negative integers b. (Recursion is not particularly natural here, but it makes for an easy warm-up exercise, much like the factorial function.)
2. Write a recursive function that prints the base 10 digits of an integer in reverse order. E.g., on input 1234, your function should print 4321.
3. Write a recursive version of binary search for sorted arrays. A plausible prototype might be `bool search(int* A, size_t n, int x);` (n is the size of A, and x is the element to search for).
4. Without referring back to these notes, write and test a function for merge sort which works on vectors (instead of arrays, as was shown here).
5. Write a version of the `perms` function which uses vectors instead of strings. Try to use the following strategy: rather than letting each element "have a turn" being first, let each have a turn being last.
6. Write a function that on input of a set S computes $\mathcal{P}(S)$, the *power set* of S. This is the set of all subsets of S. For example, if the input was the set $\{0, 1\}$, the output would be the set of sets $\{\{\}, \{0\}, \{1\}, \{0, 1\}\}$.
7. You may have noticed that the recursion trees for merge sort and our Fibonacci sequence have essentially the same shape, yet their performance characteristics were radically different. While merge sort was touted as a highly efficient procedure, the Fibonacci code was criticized as terribly inefficient. How do you explain this?
8. Show that our gcd algorithm is actually quite efficient. In particular, show that on input (a, b), it will make at most $c \cdot \log |b|$ recursive calls for some small constant c. *Hint:* focus on the second parameter (which we used to define the "size" of an instance), and then think about how large the remainder $a\%b$ could be. Sure, it might be close to b, but in that case *the next remainder won't be.* Also fun is to think about what happens when you run our algorithm on adjacent terms of the Fibonacci sequence...
9. Prove that the definition in equation (2) is equivalent to the "literal" definition of the greatest common divisor (the largest positive integer dividing both parameters). If you need a hint, try the following outline:
   (a) Let $d = \min_{\substack{x,y \in \mathbb{Z} \\ xa+yb \neq 0}} \{|xa + yb|\}$. Show that $\{xa + yb \mid x, y \in \mathbb{Z}\} = \{dz \mid z \in \mathbb{Z}\}$. (Hint: use the *division algorithm.*)
   (b) Next observe that if d' is any common divisor of a, b, then d' divides every integer linear combination $xa + yb$. In particular, $d' \mid d$.
   (c) Conclude that $d = \gcd(a, b)$.

[9] If we made our base case only the empty string instead of strings of length 0 or 1, the arrows would have been explicit.

10. **Bonus exercise**: to draw the recursion trees in these lecture notes, I wrote the following macros (modulo slightly different styling):

```
#define RTRACE_BEGIN(format,...) \
    static size_t __id=0; \
    static size_t __p=-1; \
    size_t __t = __id++; \
    if (__p == (size_t)-1) { \
        fprintf(stderr,"digraph rtree {\n"); \
        fprintf(stderr,"  edge [color=black fontcolor=black];\n"); \
        fprintf(stderr,"  node [style=filled shape=circle];\n"); \
    } \
    fprintf(stderr,"%lu [label=\"" format "\"]\n",__t,__VA_ARGS__); \
    if (__p != (size_t)-1) fprintf(stderr,"%lu -> %lu\n",__p,__t); \
    size_t __op = __p; /* save old value */ \
    __p = __t;

#define RTRACE_RET(x) \
{ \
    auto __rtrace_rv = x; \
    __p = __op; /* restore for next call */ \
    if (__p == (size_t)-1) { \
        fprintf(stderr,"}\n"); \
    } \
    return __rtrace_rv; \
}
```

Then with minor modifications to an existing recursive function, you can obtain a version that prints a representation of its own recursion tree when called. For example, our old gcd function:

```
unsigned int gcd(unsigned int a, unsigned int b)
{
    if (b == 0) return a;
    return gcd(b,a%b);
}
```

becomes the following:

```
unsigned int gcd(unsigned int a, unsigned int b)
{
    RTRACE_BEGIN("%u,%u",a,b); /* setup formatting for nodes */
    if (b == 0) RTRACE_RET(a); /* replace return with RTRACE_RET */
    RTRACE_RET(gcd(b,a%b));    /* replace return with RTRACE_RET */
}
```

Now if you evaluate gcd(15, 27), the following will be printed to `stderr` (formatting tweaked a bit for readability):

```
digraph rtree {
  edge [color=black fontcolor=black];
  node [style=filled shape=circle];
  0 [label="15,27"]
  1 [label="27,15"]
  0 -> 1
  2 [label="15,12"]
  1 -> 2
  3 [label="12,3"]
  2 -> 3
  4 [label="3,0"]
  3 -> 4
}
```

This is readable by the `dot` program from GraphViz, which is installed on your virtual machine. If your compiled program is named `myprogram` then the following command would produce an svg image of the call graph (note that we swap `stdout` with `stderr` to send only the graph description to `dot`):

```
./myprogram 3>&2 2>&1 1>&3 | dot -Tsvg -o /tmp/call-graph.svg
```

Finally, the exercise:

(a) Try to use the above to draw recursion trees of your own functions.

(b) Try to reverse engineer all this. You will need to know about the `static` keyword, about the awful `auto` type in C++,[10] and a little about format strings and preprocessor macros. Also note that these macros won't work for every situation, but they do cover quite a lot of cases.

[10] I would discourage using this feature in general. The writing of these macros is actually the *only* time I've ever used `auto`.