Lecture Summary

In this lecture, we continued our exploration of recursion. First, we saw a much more elaborate example, namely, a solution for the $n$-queens problem. Then, we saw several examples of recursive definitions, which will play an important part throughout the semester.

1 The $n$-Queens problem

The first two examples we saw of recursion in class were pretty easily coded with a simple loop, as you saw. In general, if you have a recursive function that uses just one recursive call to itself, it is often easily replaced by a loop. Recursion becomes much more powerful (as in: it lets you write short and elegant code for problems that would be much more messy to solve without recursion) when the function calls itself multiple times.

We will see several examples of that later in class with some clever recursive sorting algorithms and others. Another very common application is via a technique called Backtracking for solving complicated problems via exhaustive search. In this class, we saw the classic $n$-queens problem as an example for this technique.

In the $n$-queens problem, you want to place $n$ queens on an $n \times n$ chessboard (square grid). Each queen occupies one square on a grid and no two queens share the same square. Two queens are attacking each other if one of them can travel horizontally, vertically, or diagonally and hit the square the other queen is on. The problem is to place the queens such that no two queens are attacking each other. For instance, what you see in Figure 1 is not a legal solution: the queens in rows 1 and 2 attack each other diagonally. (All other pairs of queens are safe, though.)

![Figure 1: Illustration of the 4-queens problem](image)

Before we try to solve the problem, we make some observations. Because queens attack each other when they are in the same row or column of the chessboard, we can phrase the problem equivalently as follows: place exactly one queen per row of the board such that no two queens are in the same column or attack each other diagonally.

We can solve this by having $n$ variables $q[i]$, one per queen. Each variable loops from 0 to $n - 1$, trying all $n$ places in which the queen could be theoretically placed. Of all those at most $n^n$ ways of placing the
queen, we check if they are legal, and output them if so. Of course, it will be more efficient if we abort a
search as soon as there is an attack between queens, since placing more queens can never fix that.
So we will have an array \( q[0 \ldots (n-1)] \) that contains the positions within the rows for each of the \( n \)
queens. Those will be a global variable. Also, we’ll have a global variable for the size \( n \) of the grid:

\[
\text{int } *q; // \text{ positions of the queens} \\
\text{int } n; // \text{ size of the grid}
\]

Then, the main search function will be of the following form:

\[
\text{void search (int row)} \\
{ \text{if (row == n) printSolution ()}; // that function shows the layout} \\
\text{else} \\
{ \text{for (q[row] = 0; q[row] < n; q[row]++)} \\
{ \text{search (row+1);}} \\
}\}
\]

That’s the general outline of most Backtracking solutions. At each point where a decision must be made,
have a loop run over all options. Then recursively call the function for the remaining choices.
Of course, so far, we don’t check anywhere to make sure that the solution is legal, i.e., no pair of queens
attack each other. To do that, we want to add an array that keeps track of which squares are safe vs. attacked
by a previously placed queen. We capture it in an array \( t \) (for “threatened”), which we can make a global
variable. (It’s a two-dimensional array, which translates to a pointer to a pointer.)

\[
\text{int **t;}
\]

Now, we need to check whether the place we’re trying to place a queen at is actually legal, and update
the available positions when we do. Instead of just keeping track whether a square is attacked by any queen,
we should actually keep track of how many queens attack it. Otherwise, if we just set a flag to \text{true}, and
two queens attack a square, when we move one of them, it will be hard to know whether the square is safe.
The more complete version of the function now looks as follows:

\[
\text{void search (int row)} \\
{ \text{if (row == n) printSolution ()}; // that function shows the layout} \\
\text{else} \\
{ \text{for (q[row] = 0; q[row] < n; q[row]++)} \\
{ \text{if (t[row][q[row]] == 0)} \\
{ \text{addToThreats (row, q[row], 1);} \\
{ \text{search (row+1);}} \\
{ \text{addToThreats (row, q[row], -1);}} \\
}\}
}
We still have to write the function `addToThreats`, which increases the number of threats for the correct squares. (In class, we just put the code directly there, rather than define a function.) The function should mark all places on the same column, and on the two diagonals below the current square. For the latter, we need to make sure not to leave the actual grid. Looking at it a little carefully, you’ll see that the following function does that:

```c
void addToThreats (int row, int column, int change)
{
    for (int j = row+1; j < n; j++)
    {
        t[j][column] += change;
        if (column+(j-row) < n) t[j][column+(j-row)] += change;
        if (column-(j-row) >= 0) t[j][column-(j-row)] += change;
    }
}
```

Finally, we need to write our `main` function that reads the size, creates the dynamic arrays, initializes them, and starts the search.

```c
int main (void)
{
    cin >> n;
    q = new int [n];
    t = new int* [n];
    for (int i = 0; i < n; i++)
    {
        t[i] = new int [n];
        for (int j = 0; j < n; j++)
            t[i][j] = 0;
    }
    search (0);
    return 0;
}
```

Notice that we should really deallocate the arrays before returning from `main` — we left that out here (creating a memory leak) to avoid overloading the code.

If you do not yet fully understand how the above solution works, try tracing its execution by hand on a 5 × 5 board, by simulating all the q[i] variables by hand. That will probably give you a good idea of backtracking.

2 Some General Comments on Recursive Functions

At a high level, there are two types of recursion: `direct` and `indirect`. Direct recursion happens when a function f calls itself. That’s what we have seen so far. Not quite as frequent, but still quite common, is indirect recursion: you have two functions f, g, and f calls g, and g calls f. There is nothing particularly deep about this distinction: we’re mentioning it here mostly so that you are familiar with the terms. If you find yourself using indirect recursion and running into compiler errors, the problem could be that one of the two function definitions has to be first, and when you define that function, the compiler does not know about the other function yet. The way around that is as follows (in the examples, we assume that our functions are from `int` to `int`, but there’s nothing special about that):

```c
int f (int n); // just a declaration of the signature (this will often go in the .h file)```
int g (int n)
{
    // insert code for g here, including calls to f
}

int f (int n)
{
    // insert code for f here, including calls to g
}

This way, when the compiler gets to the definition of g, it already knows that there is a function f; when it gets to f, you have already defined g.

Another thing to keep in mind is that there are some programming languages (called functional languages) in which typically all problems are solved using recursion. Several of them do not even have loops. Some examples of such languages are ML (or its variant OCAML), Lisp, Scheme, Haskell, Gofer. There are others. Some of these (in particular, ML) are actually used in industry, and Lisp is used in the Emacs editor.

Functional languages make functions much more central than procedural ones. It is very typical to write a function that takes another function as an argument. For instance, you may think of a function g that operates on an array or list, and gets passed another function f as an argument, and what it does is apply f to each element of the array/list. (For instance, you could have a function that turns each entry of an array into a string.) This operation is called map. Another common thing is to have a function h that applies some other function to compute a single output from an entire array/list. An example would be to sum up all elements of an array/list, or to compute the maximum. You implemented those in Homework 1, and probably noticed that the code was almost the same. This operation is called reduce.

Programs that are written by applying only these two types of operations can often be very easily parallelized over large computation clusters, which is why the Map-Reduce framework has become quite popular lately (e.g., in Google’s Hadoop). It has led to a resurgence in interest in some aspects of functional programming.

From a practical perspective, when you write functional programs, it often takes longer to get the program to compile, because many logical mistakes that lead to weird behavior in C++ can’t even be properly implemented in a functional language. Once a functional program compiles correctly, it is much more often bug-free than a procedural program (assuming both are written by fairly experienced programmers).

### 3 Recursive Definitions

So far, we have talked about recursion as a programming technique. An almost equally important application of recursion is as a way of specifying objects concisely, by virtue of recursive definitions. These will come in very handy later on when defining lists, stacks, heaps, trees, and others. To be ready for that when we need it, we’ll practice here with a few easier recursive definitions. The first of these are examples that you can define pretty easily without recursion (just like our earlier examples of using recursion as a programming technique), while the later ones may be more involved (and would be very hard to define non-recursively).

1. A string of (lower-case) letters is either: (1) the empty string (often written as ε or λ), or (2) a letter ‘a’–‘z’, followed by a string of letters.

   The recursion happens in case (2), and case (1) is the base case. Of course, for this one, we could just have said that a string is a sequence of 0 or more lower-case letters, which would have been just fine. But we’re practicing recursion on easy examples here.

2. A non-negative integer is either: (1) the number 0, or (2) n + 1, where n is a non-negative integer.
Here, defining \textit{what exactly} integers are is without referring to integers in the first place may be a little puzzling. Recursion helps with that. It says that there is a first one (the number 0), and a way to get from one to the next one. In this sense, 4 is really just shorthand for $0 + 1 + 1 + 1 + 1$.

3. A palindrome is either: (1) the empty string $\epsilon$, or (2) a single letter ‘a’–‘z’, or (3) a string $xP$, where $x$ is a single letter ‘a’–‘z’, and $P$ is a palindrome itself.

Here, we needed two base cases; case (3) is the recursion. Notice that the other definition of a palindrome, “a string that reads the same forward as backward”, is correct, but much more procedural: it tells us how to test whether something is a palindrome (“Is it the same forward as backward?”), but it doesn’t tell us how to describe all of them.

4. A simple algebraic expression consists of numbers, variables, parentheses, and + and *. (We leave out - and / to keep this a little shorter.) We’ll use abundant parentheses and forget about the precedence order here. We now want to express that something like “(5*(3+x))” is legal, while “x ( 5 * * +)” is not. We can recursively say that the following are legal expressions:

- Any number. (This is a base case, and we could use our definitions of numbers above.)
- Any variable. (This is another base case; we could use our definition of strings.)
- $(A + B)$, where both $A$ and $B$ are legal expressions themselves.
- $(A * B)$, where both $A$ and $B$ are legal expressions themselves.

For this example, you’d probably be very hard-pressed to come up with a non-recursive definition. What we have written down here is called a “context-free grammar” (or CFG). There are tools (a program called \textit{bison}, which is the newer version of one called \textit{yacc}) which, given such a recursive definition, will automatically generate C code for parsing inputs that conform to the definition. They are quite useful if you are trying to define your own input format or programming language.

5. In fact, following up on the previous discussion, you can write down a complete recursive definition of the C or C++ programming language. In fact, that is how programming languages are specified. It would be pretty hopeless to try this without recursion.