Contents

More on Recursion

The code for this lecture is available in this archive[1].

Re-Introduction

We previously defined recursion as follows:

> a method is *recursive* if it calls itself.

Applied very strictly, the simplest (and most likely shortest) recursive method is the following:

```
// Warning: dangerous function!
void R()
{
  R();
}
```

It is a method (R) that simply … calls itself. Even if this method does not "do" anything, calling it will most likely make your program crash, since R will keep calling itself forever: this is actually an example of an infinite loop, and the basics of the "fork bomb" attack[2][3].

A better definition of recursion would include something about the method eventually terminating, like the following:

```
void CountDown(int n)
{
  if (n == 0)
  {
    Console.WriteLine($"{n}: Blast off!");
  }
  else
```

[1]https:/princomp.github.io/code/projects/AdvancedRecursion.zip
[2]https://en.wikipedia.org/wiki/Fork_bomb
[3]Except that the fork bomb calls itself *twice*, and in parallel.

```
    {
      Console.Write($"{n}…");
      CountDown(n - 1);
    }
  }
```

In that case, if we call e.g., `CountDown(10)`, then would be displayed:

```
10…9…8…7…6…5…4…3…2…1…0: Blast off!
```

But note that this method is not *always* terminating: indeed, calling `CountDown(-1)` actually loops forever, since removing 1 to -1 repetitively will never make it reach 0 (if we forget about overflows for an instant).

A possible way to patch this would be to have two additional method: one to count "up" to 0, and one that decides which method to call:

```
void CountUp(int n)
{
  if (n == 0)
  {
    Console.WriteLine($"{n}: Blast off!");
  }
  else
  {
    Console.Write($"{n}…");
    CountUp(n + 1); // <- Only change.
  }
}
void Count(int n)
{
  if (n < 0)
    CountUp(n);
  else
    CountDown(n);
}
Count(10);
Count(-10);
```

As we can see, `Count` itself is *not* recursive, but it calls a recursive method.

Finally, methods can be *mutually recursive*: a method `MyTurn` can call a `YourTurn` method that itself calls `MyTurn`. While neither method are recursive, they create a recursive situation, as exemplified below:

```
void MyTurn(int n)
{
  if (n < 0)
  {
    Console.WriteLine("The Game is over.");
  }
  else
  {
    Console.WriteLine("It's my turn");
    n--;
    if (n < 0)
    {
      Console.WriteLine("The Game is over.");
    }
    else
    {
      YourTurn(n);
    }
  }
}
void YourTurn(int n)
{
  Console.WriteLine("It's your turn.");
  MyTurn(n);
}
```

Note that determining how many time both methods will be executed may not be easy: in our example, if `MyTurn(4)` is called, can you determine what will be displayed?

Arrays and Recursion

Any structure over which we can iterate can be treated using recursion, and arrays are no exception. In the following, we will re-implement two simple methods using recursion: one to decide if an array is sorted, and one that implements binary search.

Sorted Array Using Recursion

Given an array and a current index, to determine if the array *is sorted*, one can:

- Make sure that the array to the left of the current index *is sorted*,
- Make sure that the value at the current index is less than the value at the next index,
- Make sure that the array to the right of the current index *is sorted*.

Note that our definition above is recursive: *being sorted* is defined using *being sorted*.

Assuming the array is sorted up to `currentIndex`, the following will return **`true`** if the rest of the array is sorted, **`false`** otherwise:

```
bool SortedH(int[] aP, int currentIndex)
{
  if (aP.Length == currentIndex + 1)
    return true;
  else if (aP[currentIndex] > aP[currentIndex + 1])
    return false;
  else
    return SortedH(aP, currentIndex + 1);
}
```

The first test check if we are done (in which case the array *is* sorted), the second compare the value at the current index with the one that follows, and the last one kicks in the reduction by stipulating that if the other two tests failed, then the array is sorted if the rest of the array is.

Compared to our informal above, we are missing the "making sure the left of the current index is sorted" bit, *unless* we start with current index … 0! Putting it all together, we can define `Sorted` calling the recursive `SortedH` method with the right arguments (and after performing some checks):

```
bool Sorted(int[] aP)
{
  if (aP == null)
    return false;
  else
    return SortedH(aP, 0);
}
```

Binary Search Using Recursion

We can perform binary search[4] using recursion:

```
bool BinFindH(int[] aP, int start, int end, int
 ↪  target)
{
  int mid = (start + end) / 2;
  if (start > end)
  {
    return false;
  }
  else
  {
    if (target == aP[mid])
    {
      return true;
```

[4]https:/princomp.github.io/lectures/collections/search#binary-search

```
      }
      else if (target > aP[mid])
      {
        return BinFindH(aP, mid + 1, end, target);
      }
      else
      {
        return BinFindH(aP, start, mid - 1, target);
      }
    }
  }
  // Binary search
  bool BinFind(int[] aP, int target)
  {
    return BinFindH(aP, 0, aP.Length - 1, target);
  }
```

Lists and Recursion

Lists are also naturally manipulated by recursive methods. We show, as an example, two ways of defining a method that construct a `string` describing a .NET list.

```
  string DisplayH(
    string retString,
    List<string> listP,
    int indexP
  )
  {
    if (listP.Count == indexP + 1)
    {
      return retString + listP[indexP] + ".\n";
    }
    else
    {
      retString += listP[indexP] + " -> ";
      return DisplayH(retString, listP, indexP + 1);
    }
  }

  string Display(List<string> ListP)
  {
    string retString = "";
    return DisplayH(retString, ListP, 0);
  }
```

Note that the `DisplayH` method is a bit cumbersome, as it must carry around

1. The whole list (`listP`),
2. The string that is being constructed (`retString`),
3. An index (`indexP`).

An alternative way of writing such a method is to

1. shorten the list as we go (using `RemoveAt`),
2. and to use a *reference* to the string,

as follows:

```
void DisplayRef(ref string descP, List<string> listP)
{
  if (listP == null || listP.Count == 0)
  {
    descP += ".\n";
  }
  else if (listP.Count == 1)
  {
    descP += listP[0] + ".\n";
  }
  else
  {
    descP += listP[0] + " -> ";
    listP.RemoveAt(0);
    DisplayRef(ref descP, listP);
  }
}
```

But note that the list is *actually* shortened by the `RemoveAt` instruction: if we additionally have to leave the original string unmodified, then a copy of the list must be created, using e.g.

```
List<string> listCopy = new List<string>(
  operatingSystems
);
```