Exercise Class 10

Contents

1 Recursion - *Blackboard AND Programming Environment*

1.1 Recursive power function

We start off by writing down a (naive) function implementing power.

```
double power(const double x, const unsigned int n) {
  if (n == 1) {
    return x;
  }

  return x*power(x, n-1);
}
```

Since a recursive function has to have a termination condition, recursion sometimes looks like induction (compare the termination condition to the anchor in induction and the recursive part to the step). However, for recursion we can also have more complicated situations like a function calling itself twice (more about that later).

The above `power` function is called exactly n times. We can represent all these function calls graphically like this:

$$\begin{tikzcd}
\boxed{n=7} \arrow[d] \\
\boxed{n=6} \arrow[d] \\
\boxed{n=5} \arrow[d] \\
\boxed{n=4} \arrow[d] \\
\boxed{n=3} \arrow[d] \\
\boxed{n=2} \arrow[d] \\
\boxed{n=1}
\end{tikzcd}$$

At this point you can have a little debate about how to improve the easy solution. If you and your students find something better than the code below, you can, of course, implement that alternative solution with them.

To improve our first attempt we use an idea that allows us to calculate x^{20} with only 5 multiplications:

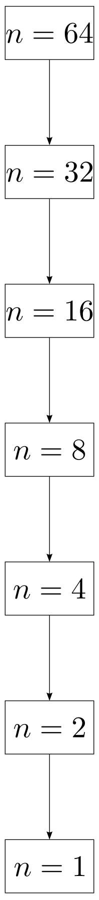
$$\begin{aligned}
x^2 &= x * x \\
x^4 &= x^2 * x^2 \\
x^8 &= x^4 * x^4 \\
x^{16} &= x^8 * x^8 \\
(x^{20} &= x^{16} * x^4)
\end{aligned}$$

Following the example of the first 4 lines of this calculation, we want to halve n with every function call. Here is a first draft.

```
double power(const double x, const unsigned int n) {
  if (n == 1) {
    return x;
  }

  double temp = power(x, n/2);
  return temp*temp;
}
```

This only works if n is a power of 2, for that we get recursive calls like this:



$$\begin{tikzcd}
\boxed{n = 64} \arrow[d] \\
\boxed{n = 32} \arrow[d] \\
\boxed{n = 16} \arrow[d] \\
\boxed{n = 8} \arrow[d] \\
\boxed{n = 4} \arrow[d] \\
\boxed{n = 2} \arrow[d] \\
\boxed{n = 1}
\end{tikzcd}$$

We conclude that for every call we halve the remaining n, resulting in $\log_2(n)+1$ function calls.

The issue with this program is that whenever n is odd, an error is introduced. This is because `n/2` is an integer division. To get rid of that problem, we use the code of our naive attempt to make sure n is even for the next function call.

```
double power(const double x, const unsigned int n) {
  if (n == 1) {
    return x;

  } else if (n % 2 == 0) {
    double temp = power(x, n/2);
    return temp * temp;

  } else {
    return x*power(x, n-1);
  }
}
```

We can easily convince ourselves that this function will work as intended. However, how many function calls do we need to calculate x^n now? The answer is that the function will call itself *at most* $2\log_2(n+1)-1$ times. That means that we still have only logarithmically many function calls, which is a lot better than our naive approach.

1.1.1 More detail*

Writing down the number of function calls for a general n is pretty hard. However, there is something else we can easily calculate and which is very useful: We can calculate the worst case.

The worst case is if n is always odd after being devided by 2. Notice that these special worst case n's are closely related to the powers of 2:

$$
\begin{aligned}
3 &= 2 * 1 + 1 (= 2^2 - 1) \\
7 &= 2 * 3 + 1 (= 2^3 - 1) \\
15 &= 2 * 7 + 1 (= 2^4 - 1) \\
31 &= 2 * 15 + 1 (= 2^5 - 1) \\
63 &= 2 * 31 + 1 (= 2^6 - 1)
\end{aligned}
$$

They are always a power of 2 minus one. So how many function calls do we need when we call our function for one of these n? Here is a graph for $n = 15$.

$$\begin{tikzcd}
\boxed{n = 15} \arrow[d] \\
\boxed{n = 14} \arrow[d] \\
\boxed{n = 7} \arrow[d] \\
\boxed{n = 6} \arrow[d] \\
\boxed{n = 3} \arrow[d] \\
\boxed{n = 2} \arrow[d] \\
\boxed{n = 1}
\end{tikzcd}$$

We conclude that we have to call the function about twice as often as for the corresponding power of two, which is $n + 1$ ($\sim 2\log_2(n+1)$) (the exact number is $2\log_2(n+1) - 1$, but we do not really care about that). The factor 2 corresponds to our function "wasting" every 2nd function call to make n even again. However, the difference between $\log_2 n$ and $2\log_2 n$ does not really matter, compared to n this is still a huge win, especially for big n.

Questions?

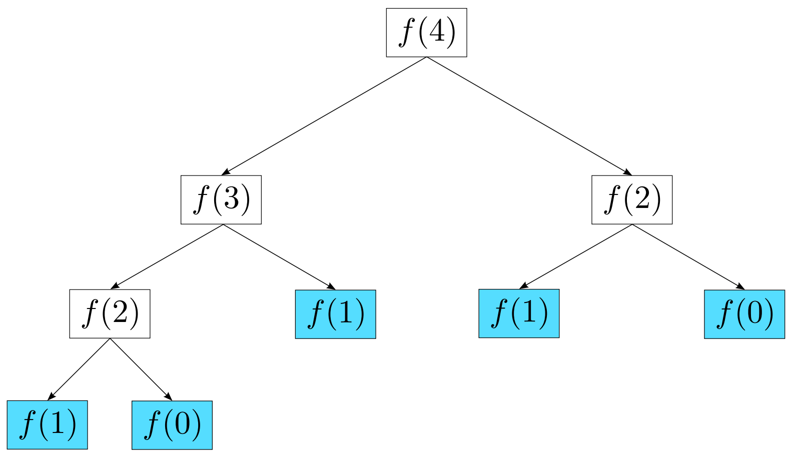
1.2 Recursion Trees - *Blackboard*

This section is for giving the students an intuitive understanding of recursion trees. That is why things will be given without proof.

In this chapter we take a look at functions which call themselves more than once in their bodies. In the lecture you already saw an example of that, namely the recursive way to calculate the nth Fibonacci number.

```
// POST: return value is the n-th
// Fibonacci number F(n)
unsigned int fib (const unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

For functions such as this one we get a recursion tree instead of a "recursion row", or whatever you want to call it. Let's have a look at the recursion tree that is caused by calling `fib(4)`.



As we can see `fib(0)` and `fib(1)` are called 5 times. It is pretty obvious why this is happening as the only way to end the recursion is calling `fib(0)` or `fib(1)`, and these always evaluate to 1 (these calls are colored blue in the figure). So we calculate the resulting Fibonacci number by adding lots of 1's in a complicated way.

However, as we see in the graph, the number of function calls (equals number of blue plus number of white boxes) is even bigger than the resulting Fibonacci number (equals number of blue boxes)! But how quickly do Fibonacci numbers grow? Well, one can show (maybe some Analysis Exercise) that for sufficiently big n they grow exponentially quickly with the golden ratio as a basis:

$$F(n) = g^n \qquad\qquad g \approx 1.6180 \tag{1}$$

So the number of function calls to calculate the nth Fibonacci number also grows at least exponentially. No wonder it already takes even a modern computer several seconds to calculate $fib(40)$ and we can not hope for it to ever finish calculating $fib(100)$. Exponential growth is very bad news indeed.

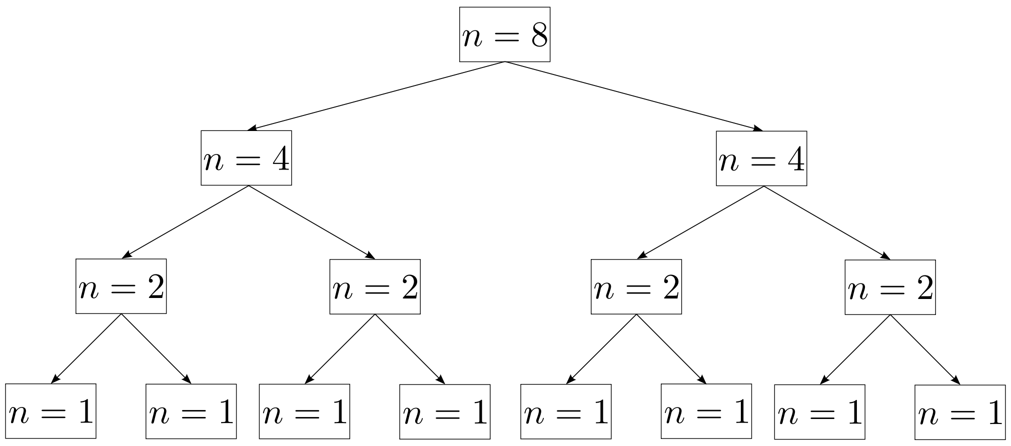
It is a good thing then, that we can do it much faster. How much faster? Aside from our bad recursive version, we have seen a much better iterative version in the lecture.

```
// POST: return value is the n-th Fibonacci number F(n)
unsigned int fib2 (const unsigned int n) {
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;           // F_1
  unsigned int b = 1;           // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a;    // F_i-2
    a = b;                      // F_i-1
    b += a_prev;                // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

So is it always bad to have a function call itself more than once like in the case of the bad Fibonacci implementation? Let us imagine a function that calls itself for $n/2$ twice.

```
unsigned int fnc (unsigned int n) {
  return fnc(n/2) + fnc(n/2);
}
```

For simplicity we assume that n is a power of 2. The resulting recursion tree looks like this.



We notice that, yes, the width of the tree still grows exponentially (1,2,4,8), however, the depth of the tree is $\log_2(n)$. Since they compensate each other we end up with

$2N - 1$ function calls. So it is not always bad for a function to call itself more than once recursively. We have to look at each individual case and estimate how fast or slow the resulting program runs. Drawing recursion trees is a nice way to visualize what the function does.

1.3 Exercises: Iterative VS Recursive Forms – *Presentation*

There are slides available. It is perfectly fine if you just cover one of them in your exercise class. Also notice that the third exercise helps the students to understand how the calculator from the lecture works. You could also do this exercise at a later point in your exercise class (for example after recursion trees). Since these programs are usually short it could be worth to write them to the blackboard.

Questions?