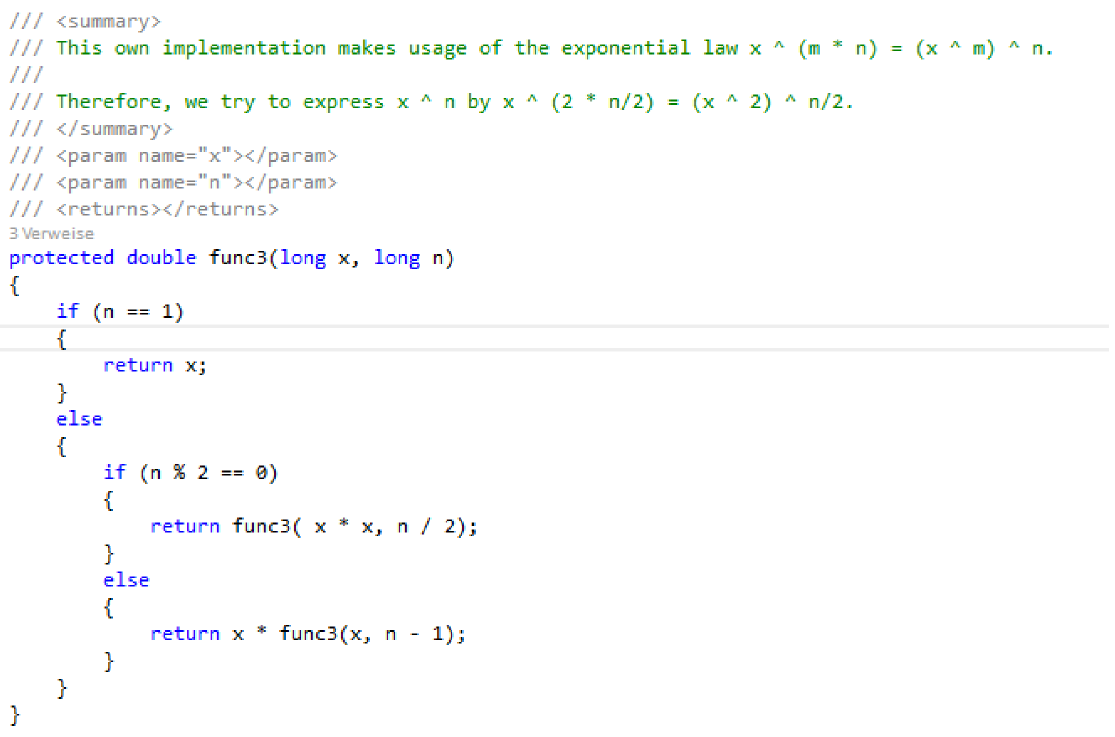
Algorithms – Homework 1

Student	Sascha Feldmann (547307)
Due Date	12-10-2014
Description	Basic Concepts: exponentiation

Task 1: Function func3()

I realized this better recursion as following:



```
/// <summary>
/// This own implementation makes usage of the exponential law x ^ (m * n) = (x ^ m) ^ n.
///
/// Therefore, we try to express x ^ n by x ^ (2 * n/2) = (x ^ 2) ^ n/2.
/// </summary>
/// <param name="x"></param>
/// <param name="n"></param>
/// <returns></returns>
3 Verweise
protected double func3(long x, long n)
{
    if (n == 1)
    {
        return x;
    }
    else
    {
        if (n % 2 == 0)
        {
            return func3( x * x, n / 2);
        }
        else
        {
            return x * func3(x, n - 1);
        }
    }
}
```

Figure 1 - Implementation of func3()

It makes use of the exponential law $x^{n*m} = (x^n)^m$ by working with the “2”-exponentials. In general, you can express the formula x^n by making use of “2”: $x^{2*\frac{n}{2}}$ if the exponent n can be divided by 2. Then, the recursive algorithm should only calculate the first part of the formula: x^2 and hand in $\frac{n}{2}$ as new parameter for n.

I will give an example now to show that the number of recursive calls can be reduced in half by the implementation of func3. I want to calculate 2^8:

- **func3(2, 8)** *256*
    - n is even, so express the exponential by using 2
- **#1: func3(2 * 2, 8 / 2) = func3(4, 4)** *256*
    - n is even, so express by using 2
- **#2: func3(4 * 4, 4 / 2) = func3(16, 2)** *256*
    - n is even, so express the exponential again by using 2
- **#3: func3(16 * 16, 2/2) = func3(256, 1)**

   - n is 1, so return x = 256

This best-case example (due the basis of 2) shows us that only 3 iterations are required. Comparing to func2, we would have needed 4 more recursive calls:

- **func2(2, 8)**
- **#1 func3(2, 7)**
- **#2 func3(2, 6)**
- **#3 func3(2, 5)**
- **#4 func3(2, 4)**
- **#5 func3(2, 3)**
- **#6 func3(2, 2)**
- **#7 func3(2,1)**

Task 2: Complexity of func3()

Let's take a look at the best case: we want to calculate the 2^8 which is a best case cause 8 = 2^3, so 8 has an integer value for the logarithm of 2.

You can identify recurrence function calls for the first iteration of func3:

$$T(n) = 1 + T(\frac{n}{2})$$

In the next iteration we will have:

$$T\left(\frac{n}{2}\right) = 1 + T(\frac{n}{4})$$

So we can express T(n) as:

$$\begin{aligned} T(n) &= 1 + T\left(\frac{n}{2}\right) \\ &= 1 + 1 + T\left(\frac{n}{4}\right) \\ &= 1 + 1 + 1 + T\left(\frac{n}{8}\right) \\ &\quad \dots \\ &= log2(k) + T\left(\frac{n}{n}\right) \end{aligned}$$

The stopping rule for $T\left(\frac{n}{n}\right)$ is n = 1. This leads to $T(1)$ and therefore we can express k by n.

$$T(n) = log2(n)$$

This means that func3(x, n) implementation is 0(log (n)).

For the worst case, we would have one more step in the recurrence formula. So, the complexity would be slightly above log2(n).

Task 3: Normalized Histogram of CPU-Time (Ticks)

Input sizes	**X**	**N**
	2	10
	2	20
	2	60
Loops for each input size	1000	
Processor	Intel Core I5-3230M	
Speed (GHz)	2.6	

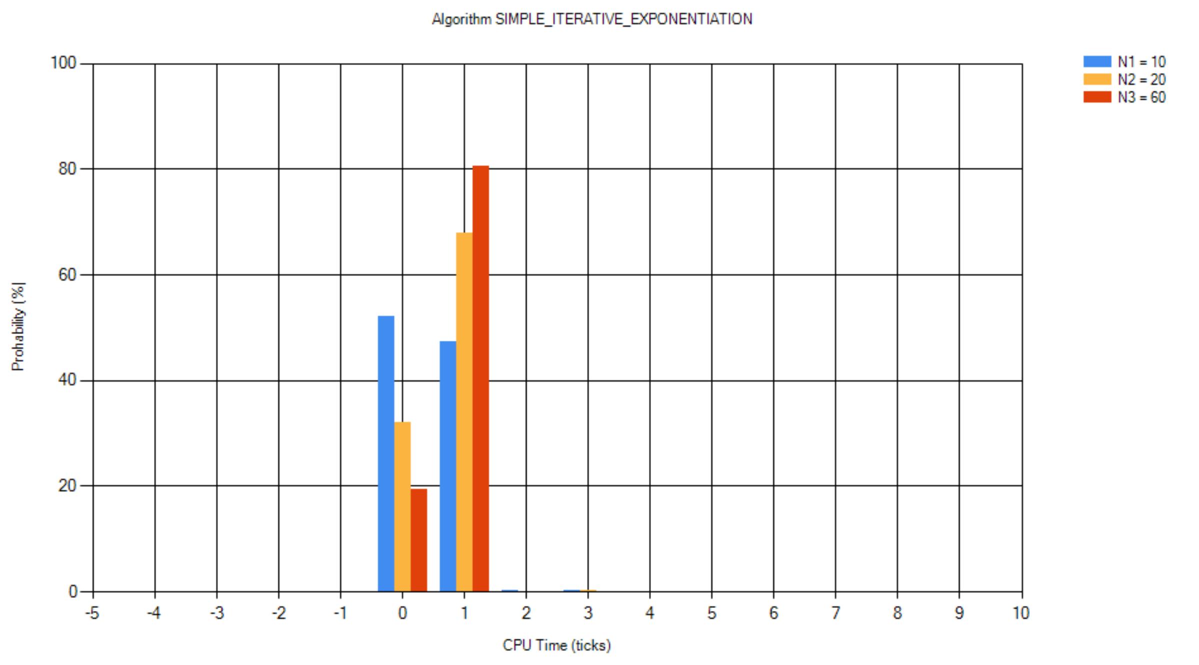


Figure 2 - Simple iterative 1

Figure 2 shows the measured CPU ticks for the calculation of x^n using the simple iterative exponentiation (func1). The smallest measured CPU ticks was 0, the biggest one 3. There were no other outside values. The low probability of the value 3 indicates that it was only the first iteration that needed 3 ticks. Afterwards, the runtime might have applied optimization.

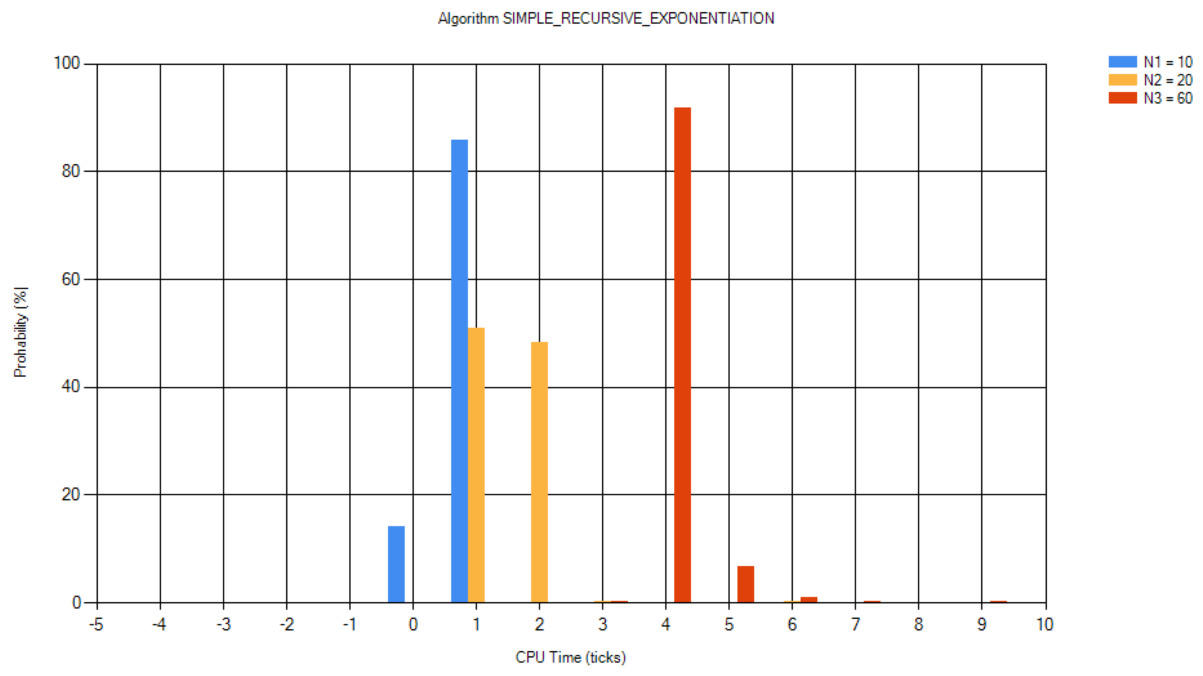


Figure 3- Simple recursive 1

Figure 3 shows higher ticks in general. The recursive algorithm func2 doesn't seem to be really efficient, because it has a probability of 90% to need 4 ticks compared to N1 which has a probability of 85% to need only 1 tick and to N2 with 50%. I think this is due the size of the function stack that is caused by the recursion. I ran another experiment with 10000 loops to check whether the runtime will optimize anything – the probabilities remain almost the same.

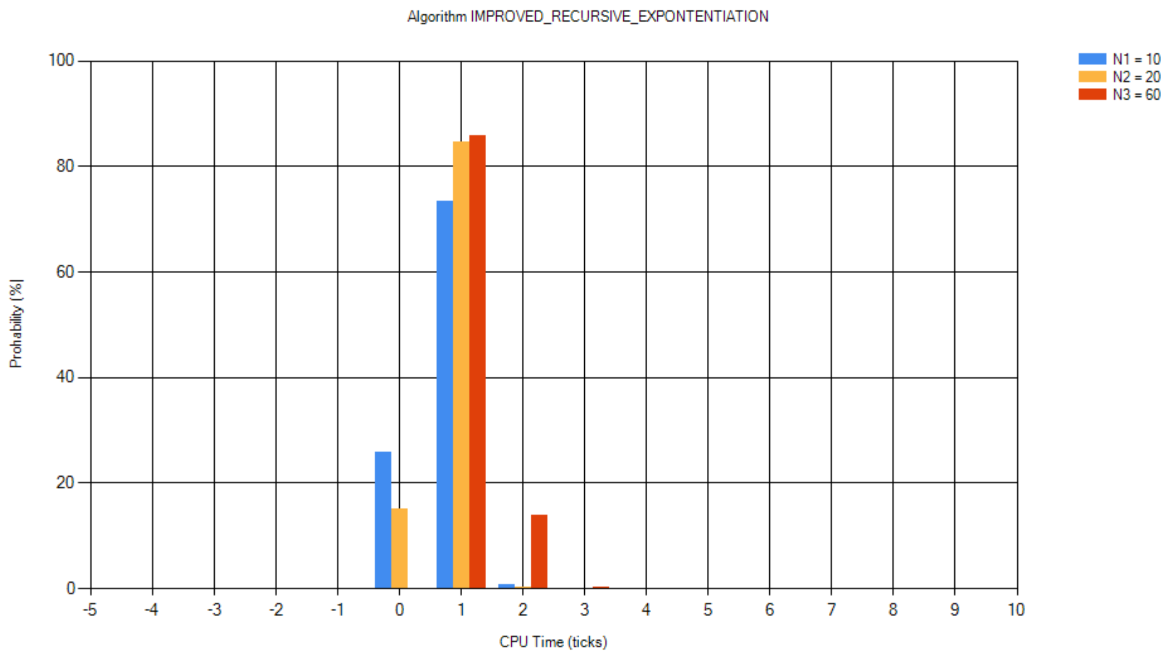


Figure 4 - Improved recursive 1

The improved recursive function shows that ticks are less than those needed in *func2()* with a high probability. The highest probability for all input sizes is 1 tick. There aren't as many higher CPU ticks as in figure 3. Compared to *func1(),* the probability to have an higher tick than 1 is higher in general. Figure 1 also shows higher probabilities to have zero CPU ticks only. In general, the improved recursive function doesn't seem to be as time-efficient than the simple iterative function *func1().*

Task 4: Comparison of means, variances, T-Tests and F-Tests

Problem Size	**Simple iterative**	**Simple recursive**	**Improved recursive**
Mean N1	0.564	1.132	1.053
Mean N2	0.412	1.391	0.678
Mean N3	0.811	4.23	1.835
Overall Mean	0.596	2.251	1.189

Table 1 – Means of algorithms and each different input size

In general you can see that the histograms above and the calculated means match; for example, the mean of the measure CPU times for the input size N3 and the improved recursive algorithm is 1.014. Figure 4 shows that the probability of N3 to be 1 is like 85% and to be 2 slightly above 15%, therefore the mean must be slightly above 1. You can also see that the means for the simple iterative algorithm seem to be the best, but the improved recursive algorithm seems to be more performant than the simple one.

Can we reject the null hypothesis for the algorithms? Let's compare the TTests (over all measured CPU times) of the algorithms in table 2.

Algorithm	**Simple recursive**	**Improved recursive**
Simple iterative	TValue: |-10.4| = 10,4 Probability (T<=T): 0 Critical TValue: 1,65 **Outcome:** TValue is larger than Critical TValue so the null hypothesis can be rejected. Also, the probability is less than the alpha value (0.05), so the null hypothesis can be rejected.	TValue: |-10.578| = 10.578 Probability (T<=T): 0 Critical TValue: 1.65 **Outcome:** Critical TValue is less than TValue, but probability is smaller than the alpha value, so the null hypothesis can be rejected.
Simple recursive	---	TValue: 8.519 Probability (T<=T): 3.886 Critical TValue: 1.65 **Outcome:** Critical TValue is less than TValue, so the null hypothesis can be rejected.

Table 2 – Results of TTest (alpha value: 0.05)

Let me continue with the FTest.

Problem Size	**Simple iterative**	**Simple recursive**	**Improved recursive**
Variance N1	8.85	9.305	11.3
Variance N2	0.869	0.48	0.359
Variance N3	0.208	7.659	0.246
Overall Variance	3.309	5.814	3.97

Table 3 –Variances of algorithms and each different input size

The variances show high values for N1 – but N1 is the lowest problem size. A reason for this can be found in the CPU times result lists. The first iteration for N1 shows a really high outstanding value for the CPU time (ticks) around 700. It seems like the runtime would do several optimizations after the iteration N1 because N2 and N3 don't show those outstanding values. So if you focus on N2 and N3, you can see that simple recursive algorithm shows higher variances than the other algorithms.

Now, let's try to reject the null hypothesis in table 4.

Algorithm	**Simple recursive**	**Improved recursive**
Simple iterative	FValue: 0.275 Probability (F<=f): 0 Critical FValue: 0.942 **Outcome:** The probability is smaller than the alpha value, so the null hypothesis can be rejected. But the critical FValue is greater than the FValue.	FValue: 1.155 Probability (F<=f): 0 Critical FValue: 0.942 **Outcome:** The probability is smaller than the alpha value, so the null hypothesis can be rejected. But the critical FValue is greater than the FValue.
Simple recursive	---	FValue: 1.126 Probability (F<=f): 0.0006 Critical FValue: 1.062 **Outcome:** The probability is smaller than the alpha value, so the null hypothesis can be rejected. But the critical FValue is greater than the FValue.

Table 4 – Results of FTest (alpha value: 0.05)

Task 5: CPU Time growth comparison

For this exercise, I first got rid of the outsider value in the simple iterative algorithm that was a result of the first iteration.

The diagram 5 shows the growth of CPU time of the 3 different algorithms for the three different problem sizes (N1=10, N2=20, N3=60). As you can see, the rate of growth for the simple iterative and improved recursive algorithms looks almost identical. The simple recursive exponentiation shows a linear growth rate, which is not as good as the logarithmic of the improved recursive algorithm.

I showed the complexity for the improved recursive algorithm in task 2. I think you can see the logarithmic growth – but the diagram is limited to N=60, which is not a good maximum value for the analysis of the growth (but higher values would lead to an overflow).

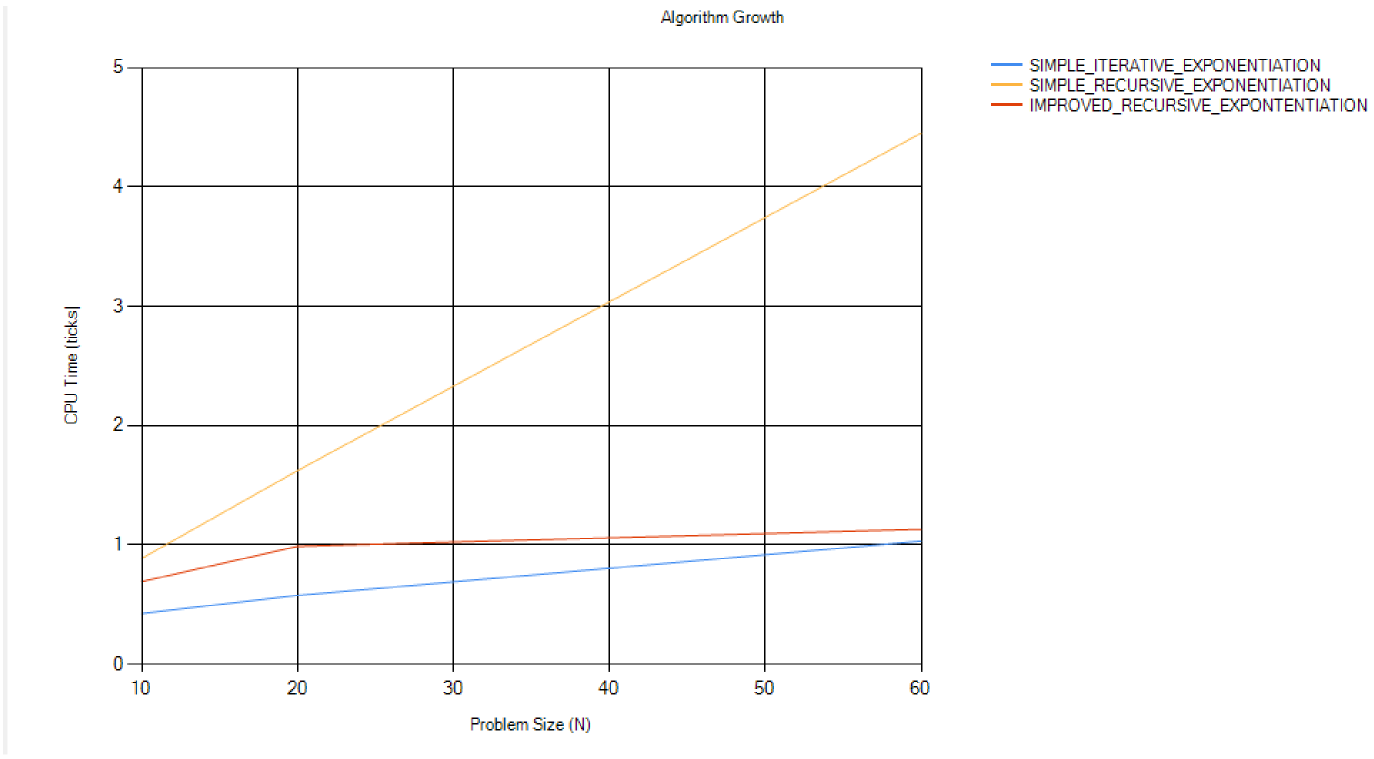


Figure 5 - CPU time Growth diagram

Problem size	Growth (N) - linear	Func(1) - Power law: $0.5 * \log(N)$	Func(2) - Power law: $0.09 * N^{0.9}$
N = 10	10	0.5 * log (10) = 0.5	0.09 * 10 ^ 0.9 = 0.71
N = 20	20	0.5 * log (20) = 0.65	0.09 * 20 ^ 0.9 = 1.334
N = 60	60	0.5 * log (60) = 0.9	0.09 * 60 ^ 0.9 = 3.586

Table 5 – Theoretical growth of func1() and func2() – simple iterative and recursive exponentiation

Table 5 shows that the power law on an expected77 linear growth applied on *func2()* mostly matches the observed values in the diagram. The growth of func1() doesn't match the expected linear growth, but follows a logarithmic growth as you can see in the diagram. This might be caused by several optimizations applied by the C# runtime – iterative algorithms seem to be optimized better than recursive ones.

Problem size	Growth (log (N))	Power law: $0.8 * \log N$
N = 10	1	0.8 * log (10) = 0.8
N = 20	1,3	0.8 * log (20) = 1,04
N = 60	1,78	0.8 * log (60) = 1,43

Table 6 – Theoretical growth of func3() – improved recursive exponentiation

You can see that the estimated values of *func3()* based on the logarithmic growth expectation mostly match the observed in the diagrams.