Task 2

2.2-3 Comparing the two algorithms for exponentiation

Algorithm-1

public static double exponentiation (double x, int n){  
 if(n==0) return 1;   
 return x \* *exponentiation*(x, n-1);  
}

Algorithm-2

public static double tweakedExponentiation (double x, int n){  
 if(n==0) return 1;  
 else if(n%2!=0){  
 return x\**tweakedExponentiation*(x\*x, (n-1)/2);  
 }  
 else return *tweakedExponentiation*(x\*x, n/2);  
}

|  |  |  |
| --- | --- | --- |
| Exponentiation | Algorithm-1 time usage | Algorithm-2 time usage |
| 1.001 ^ 1000 | 0.0023 ms | 3.457 \*10^-5 ms |
| 1.001 ^ 2000 | 0.0050 ms | 3.905 \*10^-5 ms |
| 1.001 ^ 3000 | 0.0081 ms | 4.033 \*10^-5 ms |
| 1.001 ^ 4000 | 0.0101 ms | 4.243 \*10^-5 ms |
| 1.001 ^ 5000 | 0.0138 ms | 4.567 \*10^-5 ms |
| 1.001^ 100 000 | StackOverflow | 5.703\*10^-5 ms |

If we analyze algorithm 1, we can see that it could be problematic with large numbers of n. This is simply because the number of recursion levels we need to go through is the same as n. This uses a lot of resources and when n is large enough, we run out of allocated call-stack memory. We also see that algorithm-1 has linear time complexity bound by Θ(n). We can confirm this by looking at the relationship of n and the time taken. When n doubles from 1000 to 2000 the time taken should also double, 0.0023\*2 = 0.0046, when we round up the expected answer, we get 0.0050. The same is true when n jumps from 2000 to 4000, since the time usage also doubles from 0.0050 to 0.01 respectively. Another way of arguing for the linearity of the time complexity is by showing that the algorithm has the time complexity **T(n-1) + 1** for n greater than 0, since the algorithm repeats itself for (n-1) number of times until n turns to 0. This is a linear function that can be simplified to Θ(n).

Algorithm 2, on the other hand has incredibly fewer levels of recursion compared to algorithm 1. This is due to the algorithm decreasing n by at least half with each recursion level. This can be written as T(n/2) + 1 for n>0, and when n can be described as a power of 2 the time complexity will be log2(n) + 1. Generally, we can say that such algorithms follow a sublinear time complexity that can be bound by Θ (log n). To confirm our analysis, we see that when n doubles, the time used doesn’t double but increases by a little amount. The last two rows prove this, as n increases 20-fold the time usage does not even increase 10-fold. One of the properties of the logarithmic function (log n) is that it starts increasing fast and loses its steep starting trajectory upwards, we see this with the difference of time usage at the start. This proves that the complexity is sublinear and incredibly time efficient compared to algorithm 1.

Comparing the exponent algorithms to the inbuilt exponent calculator in Java (Math.Pow)

|  |  |  |  |
| --- | --- | --- | --- |
| Exponentiation | Algorithm-1 time usage in | Algorithm-2 time usage | *Math.Pow* time usage |
| 1.001^2000 | 0.004 ms | 2.64\*10^-5 ms | 3.69\*10^-5 ms |
| 1.001^4000 | 0.008 ms | 2.74\*10^-5 ms | 3.74\*10^-5 ms |
| 1.001^6000 | 0.012 ms | 2.82\*10^-5 ms | 4.02\*10^-5 ms |
| 1.001^8000 | 0.017 ms | 2.43\*10^-5 ms | 4.19\*10^-5 ms |
| 1.001^10000 | 0.022 ms | 2.92\*10^-5 ms | 4.238\*10^-5 ms |

As we see displayed on the table, Algorithm-2 is the fastest algorithm for calculating with exponents, even beating the inbuilt exponent calculator for java. This is likely due to various data checks offered by Math.Pow in java that make it slightly slower than our recursive algorithm. Both algorithm-2 and Math.Pow are sublinear, and this can be shown with the same argumentation used in task 2.2-3, when n doubles time complexity is less than the double of the time complexity, thus proving it is sublinear. This means both algorithms, Algorithm 2 and Math.Pow are lower and upper bound by Θ(log n).