Combinatorics Project 3.
Student ID: **2018280070** Name: **Peter Garamvoelgyi**

1. Problem statement

In this assignment, we were asked to implement an algorithm for finding partition numbers. The partition number of the non-negative integer n is the number of ways it can be partitioned into an unordered sequence of positive integers s.t. these numbers sum up to n.

$$p(n) = |\{(a_1, \dots, a_n)\}| \quad \text{s.t.} \quad \begin{cases} n \in \mathbb{Z}^+, a_i \in \mathbb{Z}^+ \\ i < j \Longrightarrow a_i \geq a_j \\ \sum a_i = n \end{cases}$$

For finding $p(n)$, we can use the following generating function

$$G(x) = (1 + x + x^2 + \cdots)(1 + x^2 + x^4 + \cdots) \dots = \prod_{k=1}^{\infty} \left(\sum_{i=0}^{\infty} x^{k \cdot i} \right)$$

Evaluating the expression above, the coefficient of x^n is $p(n)$.

2. Implementation

In this section, I am going to give a brief overview of the implemented algorithms.

Given the resource constraints of ordinary computers, evaluating infinite sums and products is generally challenging. Lazy evaluation in languages like Haskell or Scala might help, but they still require a careful analysis of the order of operations.

To avoid dealing with infinite sequences, we can use the following insight: When we are looking for the coefficient of x^n, it is sufficient to evaluate the sequences only up to x^n. Terms with exponents larger than n will certainly not contribute to the coefficient of x^n:

$$m > n \quad \Longrightarrow \quad x^m \cdot y(x) = c \cdot x^k \quad k > n$$

For the same reason, it is unnecessary to include Taylor-expansions like $1 + x^m + \cdots$ when $m > n$. The intuition is that partitions of n will not contain numbers greater than n.

Applying these simplifications, our generating function becomes

$$\hat{G}(x) = (1 + x + \cdots + x^n)\left(1 + x^2 + \cdots + x^{2 \cdot \left\lfloor \frac{n}{2} \right\rfloor}\right) \dots = \prod_{k=1}^{n} \left(\sum_{i=0}^{\left\lfloor \frac{n}{k} \right\rfloor} x^{k \cdot i} \right)$$

For instance,

$$\hat{G}(4) = (1 + x + x^2 + x^3 + x^4)(1 + x^2 + x^4)(1 + x^3)(1 + x^4) = 1 + \cdots + p(4) \cdot x^4$$

For implementing this in C++, I first implemented a template class `Polynomial<T>` that encapsulates a polynomial represented using its coefficients, and allows us to multiply such polynomials. Having this class, implementation becomes fairly straightforward.

```
// generate 1 + x^d + x^2d + ... up to x^order
template <typename T>
Polynomial<T> taylor(int order, int d)
{
    // ...
}

template <typename T>
T p(int n)
{
    Polynomial<T> x = taylor<T>(n, 1);

    for (int ii = 2; ii <= n; ++ii)
    {
        x *= taylor<T>(n, ii);
        x = x.drop_after(n);
    }

    return x[n];
}

int main()
{
    std::cout << p<unsigned long long>(416) << std::endl;
    return 0;
}
```

3. Extending for large numbers

While the implementation above works with numbers up to `416`, we encounter integer overflows while attempting to apply it to larger numbers. This is due to the inherent limitation in the range of numbers representable by these data types (64 bits on my machine).

For representing larger numbers, we need a big integer library. As the implementation of these include numerous intricacies (e.g. efficient multiplication algorithms), I decided to use an open source library instead. I tried `ttmath`, `gmpxx`, `mpirxx`, and some others, and decided to use the first one, as it seemed to provide the best performance.

Using a C++ class template turned out to be a good idea: Switching from built-in integer types to `ttmath`'s integer type was almost seamless.

```
int main()
{
    std::cout << p<ttmath::UInt<2>>(1000) << std::endl;
}
```

While this solution worked well, I was still unable to find solutions for larger numbers like 4000. One way to speed things up would be to implement a more efficient polynomial multiplication algorithm instead of my trivial $O(n^2)$ implementation. Another way is parallelization.

I decided to use OpenMP for simple parallelization. This library allows us to use multiple CPU cores and threads with very low overhead in code complexity. The idea was to distribute the polynomial multiplications among the CPU's cores and then combine the results.

```
template <typename T>
T p_parallel(int n)
{
    // initialize ...

    #pragma omp parallel
    {
        int id    = omp_get_thread_num();
        int total = omp_get_num_threads();

        int from = (n / total) * id;
        int to   = (n / total) * (id + 1);

        // calculate subresult between `from` and `to` ...

        #pragma omp critical
        subresults.push_back(sub);
    }

    // combine results ...

    return x[n];
}

int main()
{
    std::cout << p_parallel<ttmath::UInt<4>>(4000) << std::endl;
    return 0;
}
```

With this extension, I was able to calculate partition numbers for numbers as large as 4000.

4. Evaluation

Below are some simple benchmarks. The system used was

```
clang version 7.0.0 (tags/RELEASE_700/final)
Target: x86_64-apple-darwin17.7.0

Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz
```

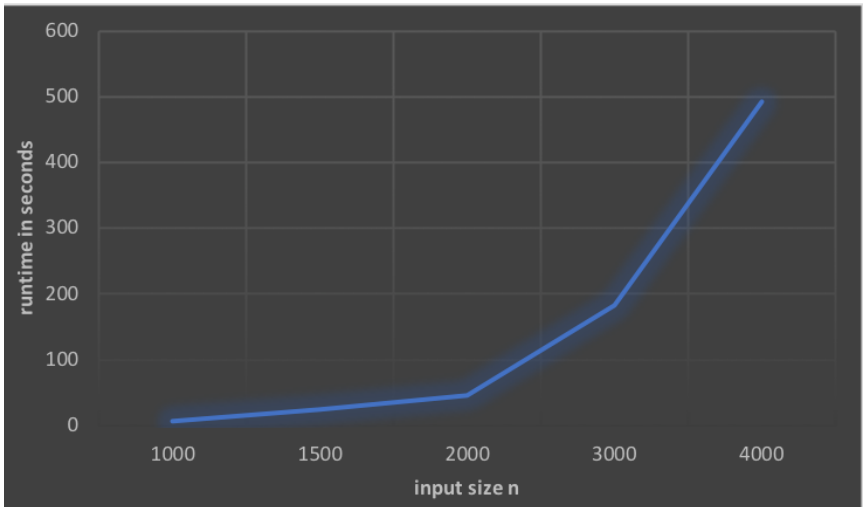
… with heavy optimization using

```
clang++ --std=c++11 -O3 -lomp -fopenmp -o main src/main.cpp
```

The results and runtimes are:

```
p(1000) = 240614678640326224736921497279 91                                    ~ 0m6.553s
p(1500) = 308614590000378498435139314406582256749                              ~ 0m23.847s
p(2000) = 472081917561941388860143240679995951220034 4166                      ~ 0m45.350s
p(3000) = 496025142797537184410324879054927095334462742231683423624           ~ 3m3.027s
p(4000) = 102415006477655137511925630791589684212249803031315091023488909389 5 ~ 8m11.776s
```

This suggests an exponential growth in the running time w.r.t. n:



A more thorough evaluation could compare different algorithms, libraries, and compilation modes, in order to assess the performance impact of these choices.

5. Conclusion

This assignment was a great opportunity for familiarizing myself with numerical computations including polynomials and big integers. It was also very rewarding to see the performance improvements after implementing each extension.

For improving performance, one could

- use a CPU with more cores,
- explore other ways of parallelization including GPUs and computing clusters,
- improve the performance of polynomial multiplication,
- improve job allocation for parallel threads,
- investigate better algorithms instead of directly calculating the polynomial product.