Graduation Project Report

Spring 2024

Hanheng He (400485161)

Project Instructor: Dr. Antoine Deza

April 10, 2024

Abstract

This report describes the condition number of a matrix, and an equivalent quantity to the condition number when entries of the matrix are specified as $\{0, 1\}$ or $\{-1, 1\}$. Based on this, the implementation of creating an $(0, 1)$ matrix is demonstrated, and the equivalent condition number is provided, showing that such kind of $(0, 1)$ matrix becomes ill-conditioned when its order grows. Furthermore, this report discusses the complexity of creating such a matrix, proving and showing exponential time complexity through an experiment.

Contents

1 Introduction

Let matrix $A \in \mathbb{R}^{n\times n}$ be an invertible matrix, with the *spectral norm* defined as $||A||_s = sup_{x\neq 0}|Ax|/|x|$, the *condition number* of A is $c(A) = ||A||_s||A^{-1}||_s$. The *condition number* is a measurement of the sensibility of the equation $Ax = b$ when the right-hand side is changed [1]. If $c(A)$ is large, then A is called $ill-conditioned$.

With such an important property of *condition number*, ill-conditioned matrices are important in numerical algebra and have been studied extensively by various researchers, such as [2], [3] and [4]. In [5], researchers restricted entries into $\{0, 1\}$ or $\{-1, 1\}$, denoted by $\mathcal{A}^1_n$ or $\mathcal{A}^2_n$, which they called $anti-Hadamard$ matrices and is of interest in linear algebra, combinatorics and related areas. With such restrictions, many quantities are equivalent to the condition number. Let A be a non-singular $(0, 1)$ matrix, $B = A^{-1} = (b_{ij})$, the following quantity as an equivalent condition number is considered in [5]:

$$\chi(A) = max_{i,j}|b_{ij}| \ and \ \chi(n) = max_A\chi(A).$$

Shown in [5], we have $c(2.274)^n \leq \chi(n) \leq 2(n/4)^{n/2}$ for some absolute positive constant c, which means $\chi(A)$ is bounded controlled by n and c. Meanwhile, since matrix A is a square matrix and only contains 0 and 1, it also stands for 0/1-polytope space, which is of great interest in geometry.

In this report, we aim to construct an ill-conditioned $(0, 1)$ matrix C satisfied

$$\chi(C) \geq 2^{\frac{1}{2}n\log n - n(2+o(1))}.$$

Hence, matrix C has a controlled infimum of its *condition number* concerning n.

Following the steps in [1], we started by generating matrix $A \in \mathcal{A}^2_n$, and we generate $B \in \mathcal{A}^1_n$ based on A. Using a series of matrix B with different shapes, we can further concatenate a $(0, 1)$ matrix C with its controlled infimum of condition number.

2 Generate Ill-conditioned Matrix C

2.1 Generate Set Ω

To start constructing matrix C with a controlled infimum of *condition number*, a set Ω with special restrictions is required. Let $|\cdot|$ denote cardinality and Δ denote symmetric different, let $m \in \mathbb{Z}^+$, $n = 2^m$. Set $\Omega = \{\alpha_0, \alpha_1, \alpha_2, ..., \alpha_n\}$ with $n + 1$ elements is required to create with restrictions that $|\alpha_i| \leq |\alpha_{i+1}|$ and $|\alpha_i\Delta\alpha_{i+1}| \leq 2$. Shown in [6], Ω is proved to exist, and we further discuss the implementational way to create such kind of set. The following description shows an implemented way.

Let $\alpha_0 = \{\varnothing\}$. We also have $\alpha_1 = \{\varnothing\}$. Suppose we have $\Omega = \{\{\varnothing\}, \{\varnothing\}, \{1\}, \{2\}, ..., \{m\}\}$ initially. Considering that every time we only take out all the sets of maximum size in Ω, and insert only one element inside in order. For example, for an existing set $\{1\}$, we insert elements one by one $2, 3, ..., m$ in order, and get a series of sets $\{1, 2\}, \{1, 3\}, ..., \{1, m\}$.

With such a way of insertion, we only need to consider if the conditions are satisfied between the last old set and the first new set, and if the continuous new sets come from different old sets. For all the sets that come from the same old set, the conditions are satisfied automatically.

```
Initially Omega = {{}, {}, {1}, {2}, ..., {m}}, insert only one element each time
    on {1}, {2}, ..., {m} in order. Make sure conditions are satisfied between:
    1. {m} and {1, _};
    2. {1, _} and {2, _}, {2, _} and {3, _}, ...
```

For the first case above, the only element we can insert to $\{1, \_\}$ is m, which is the first element if we revert the ordered insertion $2, 3, ..., m$. For the second case, let's say we have set α with size k and β with size $k-1$, if $\alpha \cup \beta = \alpha$, we can insert any element; if $\alpha \cup \beta \neq \alpha$, we can only insert an element $r \in \alpha$. Since we always insert elements in order or reversed order, if the first element of the insertion list is not in α, the last element of the insertion list or the first of the reverted list must be in α.

The following pseudo-code shows the way to generate Ω:

Algorithm 1 Generate Ω

```
Input: $m$
Output: $\Omega$
1: $\Omega \leftarrow \{\{\}, \{\}, \{1\}, \{2\}, ..., \{m\}\}$
2: for $i \leftarrow 1$ to $m-1$ do
3:     $\Phi \leftarrow$ sets in $\Omega$ with size equals $i$
4:     for $\phi$ in $\Phi$ do
5:         $\alpha \leftarrow$ [max($\phi$) + 1, ..., m]
6:         if $\alpha$ not in $\Omega$[-1] then:
7:             $\alpha \leftarrow \alpha$.reverse()
8:         end if
9:         for $a$ in $\alpha$ do
10:            $\Omega \leftarrow \{\Omega.., \{\phi.., a\}\}$        ▷ $\phi$.. means extend set $\phi$
11:        end for
12:    end for
13: end for
14: return $\Omega$
```

when $m = 4$, set Ω is shown below:

```
Omega = {{}, {}, {1}, {2}, {3}, {4},
       {1, 4}, {1, 3}, {1, 2}, {2, 4}, {2, 3}, {3, 4},
       {1, 3, 4}, {1, 2, 3}, {1, 2, 4}, {2, 3, 4},
       {1, 2, 3, 4}}
```

2.2 Construct Matrix $A \in \mathcal{A}_n^2$ with Set Ω

Shown in [1], with set $\Omega = \{\alpha_0, \alpha_1, \alpha_2, ..., \alpha_n\}$ satisfying $|\alpha_i| \leq |\alpha_{i+1}|$ and $|\alpha_i \Delta \alpha_{i+1}| \leq 2$, matrix $A \in \mathcal{A}_n^2$ can be constructed as follows such that $\chi(A) = 2^{\frac{1}{2} n \log n - n(1+o(1))}$:

For every $1 \le i, j \le n$:

$$a_{ij} = \begin{cases} -1, \ \alpha_j \cap (\alpha_{i-1} \cup \alpha_i) = \alpha_{i-1} \Delta \alpha_i \ \textit{and} \ |\alpha_{i-1} \Delta \alpha_i| = 2 \\ (-1)^{|\alpha_{i-1} \cap \alpha_j| + 1}, \ \alpha_j \cap (\alpha_{i-1} \cup \alpha_i) \neq \varnothing \ \textit{but does not meet the condition above} \\ 1, \ \alpha_j \cap (\alpha_{i-1} \cup \alpha_i) = \varnothing \end{cases} .$$

With the Ω constructed shown in section 2.1, matrix $A \in \mathcal{A}_n^2$ is shown below:

$$A = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 \\
1 & 1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & -1 & 1 & 1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 \\
1 & 1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 \\
1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & 1 & 1 & 1 \\
1 & 1 & 1 & -1 & 1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 \\
1 & 1 & -1 & 1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 \\
1 & 1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & -1 \\
1 & 1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 \\
1 & 1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & -1 & 1 & 1 & 1 & 1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & -1 & 1 \\
1 & 1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & 1 \\
1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 \\
1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1
\end{bmatrix}.$$

In [1], researchers further discussed some properties of the $(-1, 1)$ matrix. Let symmetric Hadamard matrix Q be an n by n matrix given by $q_{ij} = (-1)^{|\alpha_i \cap \alpha_j|}$, that is $Q^2 = nI_n$. There existed a lower triangular matrix L built by Ω satisfying $A = LQ$. In this report, we show $L = AQ^{-1}$ is a lower triangular matrix.

With matrix A shown above, matrix Q and L is shown below:

$$Q = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 \\
1 & 1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & -1 & 1 & 1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 \\
1 & 1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 \\
1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 \\
1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 \\
1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\
1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 & 1 & 1 & -1 \\
1 & -1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & 1 & -1 \\
1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 \\
1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 1
\end{bmatrix},$$

$$Q \times Q = \begin{bmatrix}
16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16
\end{bmatrix},$$

$$L = \begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.5 & 0.5 & 0.0 & 0.0 & -0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.5 & 0.5 & -0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.5 & 0.5 & 0.0 & 0.0 & -0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.5 & 0.0 & 0.0 & 0.5 & 0.0 & 0.0 & -0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.5 & 0.5 & 0.0 & 0.0 & 0.0 & -0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.5 & 0.0 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & -0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.25 & 0.25 & 0.0 & 0.25 & 0.25 & 0.25 & 0.25 & 0.0 & 0.0 & 0.0 & -0.75 & 0.25 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.25 & 0.0 & 0.25 & 0.25 & 0.0 & 0.25 & 0.0 & 0.25 & 0.25 & -0.75 & 0.25 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.25 & 0.25 & 0.25 & 0.25 & 0.0 & 0.25 & 0.25 & 0.0 & 0.0 & -0.75 & 0.25 & 0.0 & 0.0 \\
0.0 & 0.25 & 0.0 & 0.25 & 0.0 & 0.25 & 0.0 & 0.25 & 0.0 & 0.25 & 0.25 & 0.0 & 0.0 & -0.75 & 0.25 & 0.0 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & -0.875 & 0.125
\end{bmatrix}.$$

2.3 Mapping Matrix $A \in \mathcal{A}_n^2$ to $B \in \mathcal{A}_{n-1}^1$

With matrix $A \in \mathcal{A}_n^2$ being constructed, a mapping can be implemented to generate a matrix $B \in \mathcal{A}_{n-1}^1$ such that $\chi(B) = 2^{\frac{1}{2}n\log n - n(1+o(1))}$[1]. Consider the map Φ which assigns to any matrix $B \in \mathcal{A}_{n-1}^1$ a matrix $\Phi(B) \in \mathcal{A}_n^2$ in the following way:

$$\Phi(B) = \begin{pmatrix} 1 & 1_{n-1} \\ -1_{n-1}^T & 2B - J_{n-1} \end{pmatrix}.$$

Clearly $\Phi(B)$ is a mapping $\mathcal{A}_n^2 \to \mathcal{A}_{n-1}^1$ with a series of linear operations, so we have the following reversing way to construct matrix $B \in \mathcal{A}_{n-1}^1$ with $A = \{a_{ij}\} \in \mathcal{A}_n^2$:

$$B = \frac{1}{2}(J_{n-1} + \{a_{ij}\}_{2\le i\le n, 2\le j\le n}).$$

Notice that in the above section, the $A \in \mathcal{A}_n^2$ we constructed has it's first column as

$$\begin{pmatrix} 1 \\ 1_{n-1}^T \end{pmatrix}.$$

Therefore, we need to negative the first column. The relation between matrix $B \in \mathcal{A}_{n-1}^1$ and $A \in \mathcal{A}_n^2$ is shown as follows:

$$B = \frac{1}{2}(J_{n-1} - \{a_{ij}\}_{2\le i\le n, 2\le j\le n}).$$

With the matrix $A \in \mathcal{A}_n^2$ constructed above, the corresponding matrix B is

$$B = \begin{bmatrix}
1&0&0&0&1&1&1&0&0&0&1&1&1&0&1 \\
0&1&0&0&0&0&1&1&1&0&0&1&1&1&1 \\
0&0&1&0&0&1&0&0&1&1&1&1&0&1&1 \\
0&0&0&1&1&0&0&1&0&1&1&0&1&1&1 \\
1&0&0&0&0&1&1&0&0&0&0&1&0&0&0 \\
0&0&1&0&1&0&0&0&1&1&1&0&1&1&1 \\
0&1&0&0&0&1&0&1&1&0&1&1&0&1&1 \\
0&0&0&1&1&0&1&0&0&1&1&1&1&0&1 \\
0&0&1&0&0&1&0&1&0&1&1&0&1&1&1 \\
0&0&0&1&1&0&0&1&1&0&0&1&1&1&1 \\
1&0&0&0&0&0&1&0&0&1&1&0&0&1&1 \\
0&1&0&0&1&1&0&1&0&1&0&1&1&1&0 \\
0&0&0&1&0&1&1&0&1&1&1&0&1&1&0 \\
0&0&1&0&1&1&1&1&0&0&1&1&0&1&0 \\
1&0&0&0&0&0&0&1&1&1&1&1&1&0&0
\end{bmatrix}.$$

Shown in [1], matrix B preserves the property of its condition number, that is $\chi(B) = 2^{\frac{1}{2}n\log n - n(1+o(1))}$.

2.4 Generate and Verify Ill-conditioned Matrix C

Before constructing matrix C, [1] shows a way of concatenation that has a special property. Let S and T be two non-singular matrices of order n_1 and n_2. Define $R = S \diamond T$ as

$$R = \begin{bmatrix} s_{11} & ... & s_{1n_1} & 0 & ... & 0 \\ s_{21} & ... & s_{2n_1} & 0 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ s_{n_1 1} & ... & s_{n_1 n_1} & 0 & ... & 0 \\ 0 & 0...0 & 1 & t_{11} & ... & t_{1n_2} \\ 0 & 0...0 & 0 & t_{21} & ... & t_{2n_2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ 0 & 0...0 & 0 & t_{n_2 1} & ... & t_{n_2 n_2} \end{bmatrix}.$$

It's also shown that R satisfies

$$\chi(S \diamond T) \geq \chi(S)\chi(S).$$

Consider a $(0,1)$ matrix $C = B_1 \diamond (B_2 \diamond (...(B_{r-1} \diamond B_r))...)$ with order $\sum_{i=1}^{r} n_i = n$. With the definition of the operator $\diamond$, we have the conclusion that

$$\chi(C) \geq \prod_{i=1}^{r} \chi(B_i) > 2^{\frac{1}{2}n \log n - n(2+o(1))}.$$

This conclusion describes the infimum of $\chi(C)$, and it is evidently that matrix C is ill-conditioned concerning n.

Since the concatenation rapidly enlarges the order of the matrix, we can not show the matrix C constructed with all the steps described in previous sections. Instead, we show the results of $\chi(C)$ with different values of n. Remind that $C = B_1 \diamond (B_2 \diamond (...(B_{r-1} \diamond B_r))...)$, where r is the largest order of all the $(0,1)$ matrices A. The results is shown below:

```
r = 2, order of C = 4, χ(C) = 1.0
r = 3, order of C = 11, χ(C) = 2.0
r = 4, order of C = 26, χ(C) = 260.0
r = 5, order of C = 57, χ(C) = 106491641548.6
```

With the result above, we can observe that as order r grows, $\chi(C)$ grows rapidly, and may go to infinity.

3 Complexity Analysis of Generating Matrix C

3.1 Time Complexity of Creating Ω

With a well-defined task of creating the matrix C, a further discussion can be approached aiming to prove the limit of an algorithm, and such a field of mathematical models and techniques for demonstrating the proving is named computational complexity analysis. Since the computational complexity of a sequence is to be measured by how fast a multitap Turing machine can work out on the sequence[7], many researchers have studied computational complexity for a long time, such as [8], [9], [10] and [11].

In this section, we discuss the complexity of constructing matrix C and further show the reason for not being able to generate matrix C with large r. Since we go through all these steps with code on a modern machine, we assume that we have enough memory and focus on the time complexity. Also, we only discuss average time complexity and refer to it as time complexity.

Let m be our input parameter. To generate set Ω, we take out one set each time and insert one element inside. The insertion operation can be considered as $O(\log r)$[12] including tree data structure insertion and rebalancing, with r stands for the size of the set. However, since we always insert the largest element, we can handle it as a list data structure which leads to an $O(1)$ ideally, and we will use the $O(1)$ for an easy estimation. We also need to take out sets in Ω with maximum size, but this can be optimized by putting all these sets with the same size in a data structure when created, which leads to an $O(1)$ time complexity. At the same time, there are 2 for-loop, one with $O(m)$ and the other includes an *combination* with at least $O(m)$ time complexity. A set reversion is also included, and we can use a reversed iterator to save our time with a $O(1)$ time complexity. As shown below, with all the assignments, set insertion, set reversion operation, and an extra for-loop, a rough time complexity estimation would be $O(1) + O(m) \times (6 \times O(1) + O(m)) = O(m^2)$.

Algorithm 2 Time Complexity of Generating Ω

```
Input: $m$
Output: $\Omega$
 1: $\Omega \leftarrow \{\{\}, \{\}, \{1\}, \{2\}, ..., \{m\}\}$            ▷ $O(1)$ assignment
 2: for $i \leftarrow 1$ to $m-1$ do                                       ▷ $O(m)$ loop
 3:     $\Phi \leftarrow$ sets in $\Omega$ with size equals $i$            ▷ $O(1)$ with special data structure
 4:     for $\phi$ in $\Phi$ do                                            ▷ $\binom{m}{i} \rightarrow$ at least $O(m)$
 5:         $\alpha \leftarrow [\max(\phi) + 1, ..., m]$                   ▷ the max(...) of an ordered set is $O(1)$
 6:         if $\alpha$ not in $\Omega$[-1] then:                          ▷ $O(1)$ set visiting
 7:             $\alpha \leftarrow \alpha$.reverse()                       ▷ $O(1)$ with reversed iterator
 8:         end if
 9:         for $a$ in $\alpha$ do                                         ▷ $m - \max(\phi) - 1$, more than $O(1)$
10:             $\Omega \leftarrow \{\Omega..., \{\phi..., a\}\}$           ▷ set insertion considered as $O(1)$
11:         end for
12:     end for
13: end for
14: return $\Omega$
```

3.2 Time Complexity of Creating $(0, 1)$ Matrix C

This section discusses the time complexity of creating the matrix C with a constructed Ω. In our implementation, we use Python and package NumPy for the steps, and hence our discussion is based on them.

To generate every entry of matrix $A \in \mathcal{A}_n^2$, visits of all elements in A are necessary. Let m stand for the size of set Ω, $n = 2^m$ be the order of matrix A, the time complexity of visiting is $O(2^{2m}) = O(2^m)$. Shown in `https://wiki.python.org/moin/TimeComplexity`, the union operation time complexity is $O(s)$ with s stands for the sum of two sets, and the intersection operation time complexity is $O(min(len(s), len(t)))$ with s and t stand for the size of two sets. Since they are linear time complexity and less than $O(m)$, we skip them in our analysis and get the ideal time complexity constructing a matrix $A \in \mathcal{A}_n^2$ of at least $O(2^m)$.

Generating matrix $B \in \mathcal{A}_{n-1}^1$ requires simple matrix operations. Let $n-1$ be the order of B, the time complexity of this part is $O((n-1)^2) = O(n^2) = O(2^m)$.

To concatenate matrix C, a series of matrix B_1, B_2, ... , B_n with orders $2^1 - 1, 2^2 - 1, ..., 2^m - 1$ are needed. The main complexity of this part is assigning values into an all-zero matrix. Therefore, with matrix B_r of order r having $O(r^2)$ time complexity of assignment, to create matrix C, we can estimate a sum up, that is $\sum_{i=1}^{m} O((2^m - 1)^2) = O(m2^{2m}) = O(m2^m) = O(2^m)$.

In conclusion, to create the $(0, 1)$ Matrix C, we need an exponential time complexity. When m grows, the time required to generate matrix C grows exponentially.

The following figure 1 shows the result of an experiment, describing the relation between the time of generating matrix C and size controller m. Rapid growth is shown as expected.

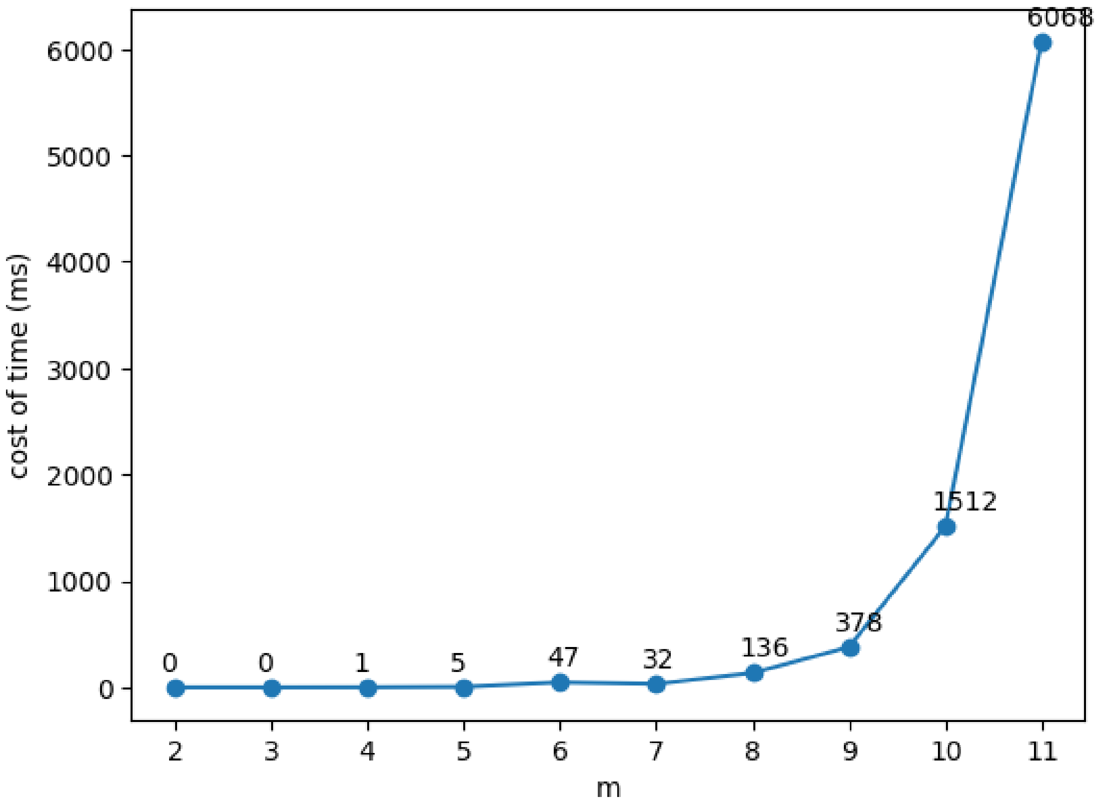


Figure 1: Cost of Time in Milliseconds Related to m

4 Conclusion

This report shows the $(0, 1)$ matrix C created with specific steps to ensure its equivalent condition number $\chi(C) > 2^{\frac{1}{2}n\log n - n(2+o(1))}$ is ill-conditioned when it's order grows. Also, we prove the exponential time complexity of creating such a matrix, showing that it is expensive to create when the order of C grows.

5 Acknowledgements

I would like to thank Antoine Deza, my supervisor, for many helpful discussions and advice. Also, I would like to thank Zhongyuan Liu and Yijun Ma for many helpful suggestions.

References

[1] Noga Alon and Văn H Vũ. Anti-hadamard matrices, coin weighing, threshold gates, and indecomposable hypergraphs. *Journal of Combinatorial Theory, Series A*, 79(1):133–160, 1997.

[2] G. H. Golub and J. H. Wilkinson. Ill-conditioned eigensystems and the computation of the jordan canonical form. *SIAM Review*, 18(4):578–619, 1976.

[3] JH Wilkinson. Note on matrices with a very ill-conditioned eigenproblem. *Numerische Mathematik*, 19:176–178, 1972.

[4] Arnold Neumaier. Solving ill-conditioned and singular linear systems: A tutorial on regularization. *SIAM review*, 40(3):636–666, 1998.

[5] R.L. Graham and N.J.A. Sloane. Anti-hadamard matrices. *Linear Algebra and its Applications*, 62:113–137, 1984.

[6] Johan Håstad. On the size of weights for threshold gates. *SIAM Journal on Discrete Mathematics*, 7(3):484–492, 1994.

[7] Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[8] Christos H. Papadimitriou. *Computational complexity*, page 260–265. John Wiley and Sons Ltd., GBR, 2003.

[9] Christos H Papadimitriou. Computational complexity. In *Encyclopedia of computer science*, pages 260–265. 2003.

[10] Ding-Zhu Du and Ker-I Ko. *Theory of computational complexity*, volume 58. John Wiley & Sons, 2011.

[11] Stephen A Cook. An overview of computational complexity. *ACM Turing award lectures*, page 1982, 2007.

[12] Pekka Kilpeläinen and Heikki Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340–356, 1995.

Appendices

Code of Implementation

In [ ]:
```
import numpy as np
from math import log

# pretty print for np.array
# from https://stackoverflow.com/questions/53126305/pretty-printing-numpy-ndarrays-using-unicode-characters/53164538#53164538
def pretty_print(A):
    if A.ndim==1:
        print(A)
    else:
        w = max([len(str(s)) for s in A])
        print(u'\u250c' + u'\u2500' * w + u'\u2510')
        for AA in A:
            print(' ', end='')
            print('[', end='')
            for i, AAA in enumerate(AA[:-1]):
                w1 = max([len(str(s)) for s in A[:, i]])
                print(str(AAA) + ' ' * (w1  - len(str(AAA)) + 1), end='')
            w1 = max([len(str(s)) for s in A[  :, -1]])
            print(str(AA[-1]) + ' ' * (w1 - len(str(AA[-1]))), end='')
            print(']')
        print(u'\u2514'+u'\u2500' * w + u'\u2518')

def compare(left: set, right: set):
    if len(left) == len(right):
        for i in range(len(left)):
            if left[i] != right[i]:
                return left[i] < right[i]
    return left < right
```

Generate $\mathcal{A}_n^2$

> Let $|\cdot|$ denotes cardinality and Δ denote symmetric different

In [ ]:
```
def Delta(left: set, right: set):
    return left.symmetric_difference(right)

def Cardi(x: set):
    return len(x)
```

> Let $n = 2^m$, Ω be a set of m element such that $|\alpha_i| \leq |\alpha_{i+1}|$ and $|\alpha_i \Delta \alpha_{i+1}| \leq 2$
> Let $\alpha_0 = \{\varnothing\}$, now genereate Ω

> utils func for checking $|\alpha_i| \leq |\alpha_{i+1}|$ and $|\alpha_i \Delta \alpha_{i+1}| \leq 2$

In [ ]:
```
def legal(alpha_i: set, alpha_i_1: set):
    return Cardi(alpha_i) <= Cardi(alpha_i_1) and Cardi(Delta(alpha_i, alpha_i_1)) <= 2

def assert_omega(ome, _m, skip=0):
    for i in range(0 + skip, len(ome) - 1):
        if not legal(ome[i], ome[i + 1]):
            print(ome[i], ome[i + 1])
        assert(legal(ome[i], ome[i + 1]))
    assert(len(ome) == 2**_m + 1)
```

In [ ]:
```
from itertools import chain, combinations

# generate Omega
def insert_one_element(initial_sets: list, super_end: int) -> list:
    res = []
    for initial_set in initial_sets:
        avail_range = list(range(max(initial_set) + 1, super_end + 1))
        if len(avail_range) == 0:
            continue
        if len(res) == 0 and avail_range[0] not in initial_sets[-1]:
            avail_range.reverse()
        elif len(res) != 0 and avail_range[0] not in res[-1]:
            avail_range.reverse()
        for avail_ele in avail_range:
            if avail_ele in initial_set:
                break
            cur_res = initial_set.copy()
            cur_res.add(avail_ele)
            res.append(cur_res)

    return res

def create_Omega(_m: int):
    initial_sets = [{i + 1} for i in range(_m)]
    grouped_res = [initial_sets]
    for _ in range(1, _m):
        grouped_res.append(insert_one_element(grouped_res[-1], _m))

    res = [set(), set()]
    for single_res in  grouped_res:
        res.extend(single_res)
    ### the follow code is for verification
    s = list(range(1, _m + 1))
    unoredered = list(chain.from_iterable(combinations(s, r) for r in range(len(s)+1)))
    for ele in unoredered:
        assert(set(ele) in res)
    assert(len(unoredered) == len(res) - 1)
    assert_omega(res, _m)
    ### verification ends
    return res
```

```
In [ ]: # test
        _test_m = 4
        _test_omega = create_Omega(_test_m)
        _test_omega
```

```
Out[ ]: [set(),
         set(),
         {1},
         {2},
         {3},
         {4},
         {1, 4},
         {1, 3},
         {1, 2},
         {2, 4},
         {2, 3},
         {3, 4},
         {1, 3, 4},
         {1, 2, 3},
         {1, 2, 4},
         {2, 3, 4},
         {1, 2, 3, 4}]
```

> now that we have $\Omega = [a_0, a_1, \ldots, a_k]$, we can generate $A = \{\alpha_{ij}\} \in \mathcal{A}_n^2$ by:
>
> $$a_{ij} = \begin{cases} -1, \ \alpha_j \bigcap (\alpha_{i-1} \bigcup \alpha_i) = \alpha_{i-1} \Delta \alpha_i \ and \ |\alpha_{i-1} \Delta \alpha_i| = 2 \\ (-1)^{|\alpha_{i-1} \bigcap \alpha_j| + 1}, \ \alpha_j \bigcap (\alpha_{i-1} \bigcup \alpha_i) \neq \varnothing \ but \ does \ not \ meet \ the \ condition \ above \\ 1, \ \alpha_j \bigcap (\alpha_{i-1} \bigcup \alpha_i) = \varnothing \end{cases}$$

```
In [ ]: def query_element(i: int, j: int, _omega: list) -> int:
            alpha_j = _omega[j + 1]
            alpha_i_1 = _omega[i]
            alpha_i = _omega[i + 1]

            if alpha_j.intersection(alpha_i_1.union(alpha_i)) == Delta(alpha_i_1, alpha_i) \
                and Cardi(Delta(alpha_i_1, alpha_i)) == 2:
                return -1
            elif Cardi(alpha_j.intersection(alpha_i_1.union(alpha_i))) != 0:
                return (-1)**(Cardi(alpha_i_1.intersection(alpha_j)) + 1)
            elif Cardi(alpha_j.intersection(alpha_i_1.union(alpha_i))) == 0:
                return 1
            else:
                raise ValueError("Undefined behavior!")

        def create_A(_omega, _m):
            A_mat = np.zeros((2**_m, 2**_m))
            for i in range(A_mat.shape[0]):
                for j in range(A_mat.shape[1]):
                    A_mat[i, j] = query_element(i, j, _omega)
            return A_mat
```

```
In [ ]: # test
        A_mat = create_A(_test_omega, _test_m)
        pretty_print(A_mat.astype(int))
```

```
[1 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 ]
[1 -1 1  1  1  -1 -1 -1 1  1  1  -1 -1 -1 1  -1]
[1 1  -1 1  1  1  1  -1 -1 -1 1  1  -1 -1 -1 -1]
[1 1  1  -1 1  1  -1 1  1  -1 -1 -1 -1 1  -1 -1]
[1 1  1  1  -1 -1 1  1  -1 1  -1 -1 1  -1 -1 -1]
[1 -1 1  1  1  1  -1 -1 1  1  1  1  -1 1  1  1 ]
[1 1  1  -1 1  -1 1  1  1  -1 -1 -1 1  -1 -1 -1]
[1 1  -1 1  1  1  -1 1  -1 -1 1  -1 -1 1  -1 -1]
[1 1  1  1  -1 -1 1  -1 1  1  -1 -1 -1 -1 1  -1]
[1 1  1  -1 1  1  -1 1  -1 1  -1 -1 1  -1 -1 -1]
[1 1  1  1  -1 -1 1  1  -1 -1 1  1  -1 -1 -1 -1]
[1 -1 1  1  1  1  1  -1 1  1  -1 -1 1  1  -1 -1]
[1 1  -1 1  1  -1 -1 1  -1 1  -1 1  -1 -1 -1 1 ]
[1 1  1  1  -1 1  -1 -1 1  -1 -1 -1 1  -1 -1 1 ]
[1 1  1  -1 1  -1 -1 -1 -1 1  1  -1 -1 1  -1 1 ]
[1 -1 1  1  1  1  1  1  -1 -1 -1 -1 -1 -1 1  1 ]
```

Check $A = LQ$

> Let Q be a n by n matrix given by $q_{ij} = (-1)^{|\alpha_i \bigcap \alpha_j|}$, Q is a symmetric Hadamard matrix, that is $Q^2 = QQ^T = nI_n$
> now we check if Q is a symmetric Hadamard matrix

```
In [ ]: def get_Q(_omega, _n):
            Q_mat = np.zeros((_n, _n))
            for i in range(_n):
                for j in range(_n):
                    Q_mat[i, j] = (-1) ** Cardi(_omega[i + 1].intersection(_omega[j + 1]))
            return np.matrix(Q_mat)

        Q_mat = get_Q(_test_omega, 2**_test_m)
        pretty_print(np.array((Q_mat).astype(int)))
        pretty_print(np.array((Q_mat * Q_mat).astype(int)))
```

```
[1 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 ]
[1 -1 1  1  1  -1 -1 -1 1  1  1  -1 -1 -1 1  -1]
[1 1  -1 1  1  1  1  -1 -1 -1 1  1  -1 -1 -1 -1]
[1 1  1  -1 1  1  -1 1  1  -1 -1 -1 -1 1  -1 -1]
[1 1  1  1  -1 -1 1  1  -1 1  -1 -1 1  -1 -1 -1]
[1 -1 1  1  -1 1  -1 -1 -1 1  -1 1  -1 1  -1 1 ]
[1 -1 1  -1 1  -1 1  -1 1  -1 -1 1  1  -1 -1 1 ]
[1 -1 -1 1  1  -1 -1 1  -1 -1 1  -1 1  1  -1 1 ]
[1 1  -1 1  -1 -1 1  -1 1  -1 -1 -1 -1 1  1  1 ]
[1 1  -1 -1 1  1  -1 -1 -1 1  -1 -1 1  -1 1  1 ]
[1 1  1  -1 -1 -1 -1 1  -1 -1 1  1  -1 -1 1  1 ]
[1 -1 1  -1 -1 1  1  -1 -1 -1 1  -1 1  1  1  -1]
[1 -1 -1 -1 1  -1 1  1  -1 1  -1 1  -1 1  1  -1]
[1 -1 -1 1  -1 1  -1 1  1  -1 -1 1  1  -1 1  -1]
[1 1  -1 -1 -1 -1 -1 -1 1  1  1  1  1  1  -1 -1]
[1 -1 -1 -1 -1 1  1  1  1  1  1  -1 -1 -1 -1 1 ]
```

```
[16 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 ]
[0  16 0  0  0  0  0  0  0  0  0  0  0  0  0  0 ]
[0  0  16 0  0  0  0  0  0  0  0  0  0  0  0  0 ]
[0  0  0  16 0  0  0  0  0  0  0  0  0  0  0  0 ]
[0  0  0  0  16 0  0  0  0  0  0  0  0  0  0  0 ]
[0  0  0  0  0  16 0  0  0  0  0  0  0  0  0  0 ]
[0  0  0  0  0  0  16 0  0  0  0  0  0  0  0  0 ]
[0  0  0  0  0  0  0  16 0  0  0  0  0  0  0  0 ]
[0  0  0  0  0  0  0  0  16 0  0  0  0  0  0  0 ]
[0  0  0  0  0  0  0  0  0  16 0  0  0  0  0  0 ]
[0  0  0  0  0  0  0  0  0  0  16 0  0  0  0  0 ]
[0  0  0  0  0  0  0  0  0  0  0  16 0  0  0  0 ]
[0  0  0  0  0  0  0  0  0  0  0  0  16 0  0  0 ]
[0  0  0  0  0  0  0  0  0  0  0  0  0  16 0  0 ]
[0  0  0  0  0  0  0  0  0  0  0  0  0  0  16 0 ]
[0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  16]
```

> And verify matrix $L = AQ^{-1}$ is a lower triangular matrix

```
In [ ]: pretty_print(np.array(np.matrix(A_mat) * np.matrix(Q_mat).I))
```

```
[1.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.0   1.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.0   0.0   1.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.0   0.0   0.0   1.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.0   0.0   0.0   0.0   1.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.5   0.5   0.0   0.0   -0.5  0.5   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.0   0.0   0.0   0.5   0.5   -0.5  0.5   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.0   0.0   0.5   0.5   0.0   0.0   -0.5  0.5   0.0   0.0   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.0   0.5   0.0   0.0   0.5   0.0   0.0   -0.5  0.5   0.0   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.0   0.0   0.0   0.5   0.5   0.0   0.0   0.0   -0.5  0.5   0.0   0.0   0.0   0.0   0.0    0.0  ]
[0.0   0.0   0.5   0.0   0.5   0.0   0.0   0.0   0.0   -0.5  0.5   0.0   0.0   0.0   0.0    0.0  ]
[0.25  0.25  0.0   0.25  0.25  0.25  0.25  0.0   0.0   0.0   -0.75 0.25  0.0   0.0   0.0    0.0  ]
[0.0   0.0   0.25  0.0   0.25  0.25  0.0   0.25  0.0   0.25  0.25  -0.75 0.25  0.0   0.0    0.0  ]
[0.0   0.0   0.0   0.25  0.25  0.25  0.25  0.0   0.25  0.25  0.0   0.0   -0.75 0.25  0.0    0.0  ]
[0.0   0.25  0.0   0.25  0.0   0.25  0.0   0.25  0.0   0.25  0.25  0.0   0.0   -0.75 0.25   0.0  ]
[0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125 -0.875 0.125]
```

Generate $\mathcal{A}^1_{n-1}$

> Let $B \in \mathcal{A}^1_{n-1}$, $\Phi(B) \in \mathcal{A}^2_n$:
>
> $$\Phi(B) = \begin{pmatrix} 1 & 1_{n-1} \\ -1^T_{n-1} & 2B - J_{n-1} \end{pmatrix}$$
>
> notice that the $A_2 \in \mathcal{A}^2_n$ we have has it's first column as:
>
> $$A_2 = \Phi(B) = \begin{pmatrix} 1 \\ 1^T_{n-1} \end{pmatrix}$$
>
> we need to multiply $\{\alpha_{ij}\}_{2 \leq i \leq n, 2 \leq j \leq n} \in \mathcal{A}^2_n$ with -1
> so we can generate matrix $A_1 \in \mathcal{A}^1_{n-1}$ with $A_2 = \{\alpha_{ij}\} \in \mathcal{A}^2_n$ using the following relation:
>
> $$A_1 = \frac{1}{2}(J_{n-1} - \{\alpha_{ij}\}_{2 \leq i \leq n, 2 \leq j \leq n})$$

```
In [ ]: def create_A_1_mat(A_2_mat):

            A_1_mat = (np.ones(A_2_mat.shape[0] - 1) - A_2_mat[1:, 1:]) * 0.5
            assert((A_1_mat.max() == 1 and A_1_mat.min() == 0) if A_1_mat.shape[0] > 1 else True)
            return A_1_mat
```

```
In [ ]: # test
        pretty_print(create_A_1_mat(A_mat).astype(int))
```

```
[1 0 0 0 1 1 1 0 0 0 1 1 1 0 1]
[0 1 0 0 0 0 1 1 1 0 0 1 1 1 1]
[0 0 1 0 0 1 0 0 1 1 1 1 0 1 1]
[0 0 0 1 1 0 0 1 0 1 1 0 1 1 1]
[1 0 0 0 0 1 1 0 0 0 0 1 0 0 0]
[0 0 1 0 1 0 0 0 1 1 1 0 1 1 1]
[0 1 0 0 0 1 0 1 1 0 1 1 0 1 1]
[0 0 0 1 1 0 1 0 0 1 1 1 1 0 1]
[0 0 1 0 0 1 0 1 0 1 1 0 1 1 1]
[0 0 0 1 1 0 0 1 1 0 0 1 1 1 1]
[1 0 0 0 0 0 1 0 0 1 1 0 0 1 1]
[0 1 0 0 1 1 0 1 0 1 0 1 1 1 0]
[0 0 0 1 0 1 1 0 1 1 1 0 1 1 0]
[0 0 1 0 1 1 1 1 0 0 1 1 0 1 0]
[1 0 0 0 0 0 0 1 1 1 1 1 1 0 0]
```

```
In [ ]: def funcX(A):
            return np.absolute(np.matrix(A).I).max()

        def get_o_1(_X, _n):
            return (0.5 * _n * log(_n, 2) - _n - log(_X, 2)) / _n
```

Verify $\chi(A) = 2^{\frac{1}{2}nlogn - n(1+o(1))}$

> with $\chi(A) = max_{i,j}|A^{-1}|$

```
In [ ]: for m in range(2, 9):
            omega = create_Omega(m)
            A_2_mat = create_A(omega, m)
            X_A_2 = funcX(A_2_mat)
            n = A_2_mat.shape[0]
            o_1 = get_o_1(X_A_2, n)
            print("m = {m_term}, n = {n_term}, χ(A_2) = {X_A_2_term}, o(1) term = {o_1_term:.3f}".format(m_term=m, n_term=n, X_A_2_term=X_A_2, o_1_term=o_1))
```

```
m = 2, n = 4, χ(A_2) = 0.5, o(1) term = 0.250
m = 3, n = 8, χ(A_2) = 1.0, o(1) term = 0.500
m = 4, n = 16, χ(A_2) = 131.50000000000142, o(1) term = 0.560
m = 5, n = 32, χ(A_2) = 2247877518.8656816, o(1) term = 0.529
m = 6, n = 64, χ(A_2) = 3344658601622888.0, o(1) term = 1.194
m = 7, n = 128, χ(A_2) = 1596000892827338.0, o(1) term = 2.105
m = 8, n = 256, χ(A_2) = 1231531590891313.5, o(1) term = 2.804
```

Verify $\chi(A') = 2^{\frac{1}{2}nlogn-n(1+o(1))}$

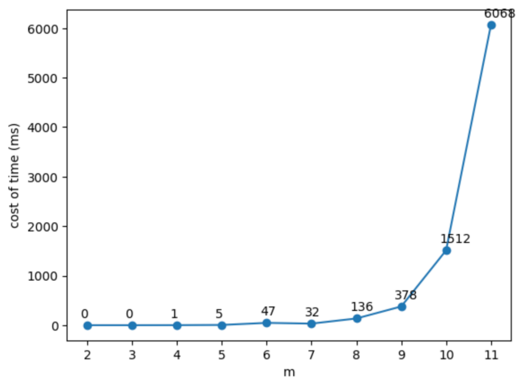
In [ ]:
```
for m in range(2, 7):
    omega = create_Omega(m)
    A_mat = create_A(omega, m)
    A_1_mat = create_A_1_mat(A_mat)
    X_A = funcX(A_1_mat)
    n = A_1_mat.shape[0] + 1
    o_1 = get_o_1(X_A, n)
    print("m = {m_term}, n = {n_term}, χ(A) = {X_A_term}, o(1) term = {o_1_term:.3f}".format(m_term=m, n_term=n, X_A_term=X_A, o_1_term=o_1))
```

```
m = 2, n = 4, χ(A) = 1.0, o(1) term = 0.000
m = 3, n = 8, χ(A) = 2.0, o(1) term = 0.375
m = 4, n = 16, χ(A) = 260.0000000000009, o(1) term = 0.499
m = 5, n = 32, χ(A) = 4495480119.287977, o(1) term = 0.498
m = 6, n = 64, χ(A) = 4212068385296025.0, o(1) term = 1.189
```

Generate matrix $C = A_1 \diamond (A_2 \diamond (\ldots(A_{r-1} \diamond A_r))\ldots)$

In [ ]:
```
import time
import matplotlib.pyplot as plt

def retangle_operator(S, T):
    top_right = np.zeros((S.shape[0], T.shape[1]),dtype=int)
    btn_left = np.zeros((T.shape[0], S.shape[1]),dtype=int)
    if btn_left.shape[1] > 0:
        btn_left[0, -1] = 1
    return np.asarray(np.bmat([[S, top_right], [btn_left, T]]))

def generate_C(_r: int):
    res = None
    for m in range(1, _r + 1):
        omega = create_Omega(m)
        A_mat = create_A(omega, m)
        A_1_mat = create_A_1_mat(A_mat)
        res = A_1_mat if res is None else retangle_operator(res, A_1_mat)
    return res

range_list = list(range(2, 12))
time_list = []

for r in range_list:
    start = time.time()
    C_mat = generate_C(r)
    # pretty_print(C_mat.astype(int))
    X_C = funcX(C_mat)
    n = C_mat.shape[0]
    o_1 = get_o_1(X_C, n)
    end = time.time()
    time_list.append((end - start) * 1000)
    print("r = {r_term}, max(n) = {n_term}, χ(C) = {X_C_term}, o(1) >= {o_1_term:.3f}".format(r_term=r, n_term=n, X_C_term=X_C, o_1_term=o_1))

plt.plot(range_list, time_list, marker='o')
plt.xlabel("m")
plt.xticks(range_list)
plt.ylabel("cost of time (ms)")
for x, y in zip(range_list, time_list):
    plt.text(x-0.15, y + 150, "%.0f" % y)
plt.show()
```

```
r = 2, max(n) = 4, χ(C) = 1.0, o(1) >= 0.000
r = 3, max(n) = 11, χ(C) = 2.0, o(1) >= 0.639
r = 4, max(n) = 26, χ(C) = 260.00000000000443, o(1) >= 1.042
r = 5, max(n) = 57, χ(C) = 106491641548.59639, o(1) >= 1.274
r = 6, max(n) = 120, χ(C) = 7.186183766512361e+24, o(1) >= 1.765
r = 7, max(n) = 247, χ(C) = 8.981448629516596e+37, o(1) >= 2.464
r = 8, max(n) = 502, χ(C) = 1.9679934451880236e+52, o(1) >= 3.140
r = 9, max(n) = 1013, χ(C) = 3.962544275444787e+64, o(1) >= 3.780
r = 10, max(n) = 2036, χ(C) = 3.4399058744893378e+75, o(1) >= 4.373
r = 11, max(n) = 4083, χ(C) = 3.2109048892035864e+86, o(1) >= 4.927
```


cost of time (ms)
m