The Josephus Problem

Five Algorithms and Comparisons

Amandeep Singh | Vasilis Valatsos
1036860 | 1037683

Project for the course
Databases & Algorithms

Università degli Studi di Torino,
Department of Mathematics ”Giuseppe Peano”,
Laurea Magistrale in Stochastics and Data Science

Torino, May 2022

Contents

List of Figures

1 | Historical Background

Near the start of the Great Jewish Revolt (66-73AD), Roman General Vespasian[1] and his son Titus[2] laid siege on the city of Yodfat, whose troops were commanded by Yosef ben-Matityahu[3]. On the 47th and last day of the siege, a deserter of the Jewish forces reached the Romans and betrayed the city. On the next day a small group of Romans led by Titus himself managed to scale the walls and open the gates to let in the entire Roman army. Taken by surprise, the Jews were furthermore confounded by a thick mist, and the Romans quickly took hold of the summit, pursuing the inhabitants down the eastern slope.

> And for the Romans, they so well remembered what they had suffered during the siege, that they spared none, nor pitied any, but drove the people down the precipice from the citadel, and slew them as they drove them down; at which time the difficulties of the place hindered those that were still able to fight from defending themselves; for as they were distressed in the narrow streets, and could not keep their feet sure along the precipice, they were overpowered with the crowd of those that came fighting them down from the citadel.
>
> *—Flavius Josephus, The Wars of the Jews, Book III, Ch. 7, sec. 34*

Meanwhile, Yosef ben-Matityahu hid in one of the many caves around the area alongside 40 other Jewish citizens. Even though Yosef was in favour of surrendering, most of the people alongside him prefered death rather than surrendering to the Romans. However, since in the Talmud suicide is considered a sin, they chose to draw lots and to kill each other. When there were only two people left, Yosef and another man, Yosef managed to convince the other person to give themselves up rather than letting one of them kill the other and then commit suicide.

> However, in this extreme distress, he was not destitute of his usual sagacity; but trusting himself to the providence of God, he put his life into hazard [in the manner following]: "And now," said he, "since it is resolved among you that you will die, come on, let us commit our mutual deaths to determination by lot. He whom the lot falls to first, let him be killed by him that hath the second lot, and thus fortune shall make its progress through us all; nor shall any of us perish by his own right hand, for it would be unfair if, when the rest are gone, somebody should repent and save himself." This proposal appeared to them to be very just; and when he had prevailed with them to determine this matter by lots, he drew one of the lots for himself also.
>
> *—Flavious Josephus, The Wars of the Jews, Book III, Ch. 8, sec. 7*

After they were captured and taken to the Roman general Vespasian, Yosef managed to earn Vespasian's favour, who spared Yosef's life, deciding to keep him as his slave and interpreter. Having fully defected to the Romans during the Great Revolt, as well as becoming a close friend of Titus, after Vespasian became Imperator of the Roman Empire, Yosef was granted his freedom and assumed the name Flavious Josephus, in honor of the family name of Vespasian and Titus.

[1] Future Imperator Caesar Vespasianus Augustus (9-79AD).
[2] Future Imperator Titus Caesar Vespasianus Augustus (39-81AD).
[3] Flavious Josephus (37-100AD)

2 | General Description of the Problem

The previous chapter raised some uniquely interesting questions. First of all, how did Josephus out of all the people survive? Was it divine providence, a stroke of luck or maybe something more? How easy would it have been for Josephus to figure out where to stand in the circle of men killing each other so that he would be one of the last two people to survive? Is there a solution such that you can always be the final survivor for an arbitrary amount of men stuck with you in a cave? Turns out, there is.

For a given number of people (n) and a given number of steps (k), the algorithm starts at the first person who looks $k-1$ places ahead[1] and kills the person at that spot. Then the process is repeated in a circle until only one person survives, who is the winner $W(n)$.

Example

Assuming that $n = 10$ and $k = 2$, we start counting from position $i = 1$ and the person at $i = 2$ is the one to die, i.e. every second person dies. Then, the person in $i = 3$ kills the person in $i = 4$ etc. until we reach $i = 9$ who kills $i = 10$ and then we loop back to $i = 1$ who repeats the process, this time killing $i = 3$. Now, after $i = 1$, $i = 2, 3, 4$ are all dead so it is $i = 5$'s turn to kill $i = 7$ and then $i = 9$, who is the last on the list, kills $i = 1$ since the topology of the problem is a circle. Finally, only $i = 5$ and $i = 9$ are alive, and it's $i = 5$'s turn who kills $i = 9$ and is the last man standing. The entire process looks like the following

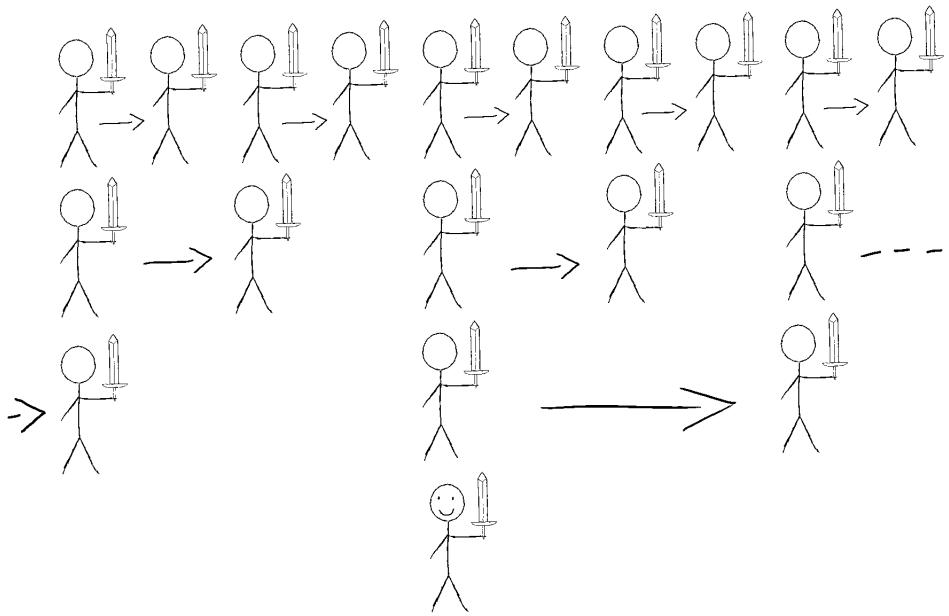

Figure 2.1: Graphic solution of the Josephus problem. For $n = 10,\ k = 2$, we have $W(n) = 5$.

[1] This is because in the counting of k people one starts counting from themselves.

From producing the above graph for various values of n, one can infer a lot of information on the process that will allow them to skip over many steps in the case where $k = 2$ which is the one that will be studied here. First of all, for arbitrary n, the first dead are the even numbers, so one should always choose an odd number to sit at in the circle. Secondly, if we write down the winning seat for $n = 1, \ldots, 16$ we might notice an interesting pattern emerging.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$W(n)$	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15	1

Table 2.1: Winning seat $(W(n))$ for each number of people (n) from $n = 1$ to $n = 16$, for $k = 2$

Every time n is a power of two, meaning it can be expressed as $n = 2^\ell$, we have that $W(2^\ell) = 1$ always, so should one find themselves in a cave with 2^ℓ number of people, they should always sit on the first seat to ensure their survival.

Using the above we can go a step further and provide a smart and elegant general solution to the problem. We know that any number can be expressed as

$$n = 2^\ell + p \tag{2.1}$$

where 2^ℓ is the power of two closest to, but not higher than, n. Some random examples are:

$$\begin{array}{lll} 5 = 2^2 + 1 & 2847 = 2^{11} + 823 & 85 = 2^6 + 21 \\ 16 = 2^4 + 0 & 566 = 2^9 + 54 & 127 = 2^6 + 63 \end{array}$$

After the beginning of the first round, after p number of steps, we can remove the dead from the circle and we can see that we have 2^ℓ number of people alive, meaning that the person that will now be the first to kill, will be the one to survive. We can formally state the above as a Proposition, which we will not prove.

Proposition

For n people, where $n = 2^\ell + p$ such that $0 \leq p \leq n$, we have that

$$W(n) = 2p + 1 \tag{2.2}$$

Finally, there is a very interesting trick that can be used to find the winning position, exactly because powers of 2 have a winning solution of 1 always. If one expresses any number in it's binary form, say for example 69,

$$69_{10} = 2^6 + 2^2 + 2^0 = 1000101_2$$

if the first digit is moved to the back

$$1000101_2 \longrightarrow 0001011_2 = 1011_2 = 11_{10}$$

we get that $W(69) = 11$.

3 | Algorithms

Armed with the ideas from the previous chapter, we are now ready to approach this problem in a plethora of ways, the first of which is the most simplistic and almost surely the most expensive in terms of time. The rest are arranged in the expected complexity, from biggest to smallest. Here, some predictions of the time complexity are included for each algorithm, but formal derivations are included in Appendix A.

3.1 Brute Force

Brute force algorithms are the most basic and fundamental algorithms, and are designed to exhaustively perform every numerical computation to simulate the process.

```
def bruteforce(n, k):
    vector = np.arange(1, n+1)
    while len(vector[vector!=0])>1:
        for i in np.arange(len(vector)-1):
            if vector[i] != 0:
                for j in np.arange(1,n):
                    if vector[i+j] != 0:
                        vector[i+j] = 0
                        break
        if vector[-1] != 0:
            vector[0] = 0
        vector = vector[vector>0]
    return int(vector[0])
```

We are able to break down the algorithm into the following steps:

- We first create a **vector** that includes the numbers from 1 up to the given n, with the intent to use the value of each element to keep track of the actual position of the winning seat.
- We then begin an iterative **whileloop** that repeats the process for as long as the length of the array is bigger than one, i.e., more than one people survive.
- Inside the **whileloop**, there is a **forloop** that runs over the entire length of **vector**, changed appropriately so the code doesn't produce an **IndexError**. The **forloop** checks whether the person who's turn it is is alive or dead (where dead is signified by the number 0), and if the person is alive he tries to find the closest non-dead person to kill, by means of a **nested forloop**.
- To simulate the cyclical topology of the problem, once the function reaches the final index of the array, it checks whether that person is alive or dead using an **if statement** since, if the person is alive, he is the one that plays and should kill the first person that is alive on the new iteration.
- The final step is at the end of each iteration, to remove the dead people, reducing the size of the array.

We expect this algorithm to be the slowest, roughly $\mathcal{O}(n^2)$ since as n grows, there is more and more distance between the living people, meaning that we have to iterate almost over the entire length of the **vector**. So we have n times $almost\ n$, because of the **nested forloop**, giving us $\sim n^2$ actions.

3.2 Recursion

The problem lends itself very easily to the implementation of a recursion algorithm, which has the following characteristics

- The base case, which is the one the function works towards to finalize,
- A set of rules that reduce successive cases towards the base case.

Below is the algorithm coded to solve the Josephus problem recursively

```
1 def recursion(n, k):
2     if n == 1:
3         return 1
4     else:
5         return (recursion(n - 1, k)+k-1)%n + 1
```

The Python program is self-descriptive but the analysis is as follows

- The base case is for $n = 1$ where obviously the last surviving person survives and is the winner.
- For $n > 1$, we have that after the death of a person, we have $n - 1$ people left, so we call **recursion(n-1)** to get the updated circle with $n-1$ people. Because of the change in the number of people, the algorithm would be wrong with its handling of the process, since it would start the $n - 1$ problem beginning from $k\%n + 1$, so we have to adjust it accordingly, so it continues from where it stopped in the original state.

Due to the recursive nature of the algorithm, we expect the complexity of this algorithm to be $\mathcal{O}(n)$.

3.3 Stacking Method

Looking back to Section 1.1, there is a clear way to optimize this process via the use of **lists** rather than **arrays**.

```
1  def stack(n, k):
2      vector=[]
3      for i in range(1,n+1):
4          vector.append(i)
5      if len(vector) == 1:
6          return 1
7      else:
8          i = 0
9          while len(vector) > 1:
10             i = (i+k-1)%len(vector)
11             vector.pop(i)
12         return vector[0]
```

- An empty **vector** is initialized, and is then populated with values from 1 to n. This method of using a **forloop** was used because the methods of **np.arange** and **range** didn't behave well with **pop()**, specifically the program returned an `AttributeError: 'range' object has no attribute 'pop'`
- In case $n = 1$, there is a condition to check the length of the **vector**. In this case, the algorithm **returns** 1.
- If $n \neq 1$, we can initiate a **whileloop** to finish the killings once one person is left alive. The algorithm begins at $i = 1$, who kills $i = k$.

- Since $i = k$ is dead, they are **popped** out of the list and the new list of size $n - 1$ restarts, this time with $i = 1$ being shifted to the next person to kill someone.

Since for an increase of n we have an equal amount of indeces of the **vector** we expect the complexity to be $\mathcal{O}(n)$.

3.4 Minimal Computation Methods

The following are not technically algorithms, being explicit solutions to the problem that are only applicable to this problem, but they are included due to their speed, efficiency, and their overall contribution to helping visualise the effects of writing a slow versus a fast algorithm. All of the following algorithms are expected to have time complexity $\mathcal{O}(1)$, due to their finite, constant amount of actions.

3.4.1 General Solution

Using formula from Eq. 2.2, it is easy to write a program that finds the nearest power of two such that $n \geq 2^{\ell}$.

```
1 def gensol(n):
2     val = n - np.power(2, np.floor(np.log(n)/np.log(2)))
3     return int((2*val)+1)
```

3.4.2 Binary Trick

In Ch. 2 we also introduced a trick, whereby writing the number n in it's binary form and transfering the first digit to the last spot we got the correct answer.

```
1 def trick(n):
2     win = list(str(bin(n)[2:]))
3     win = np.append(win, win[0])
4     win = np.delete(win, 0)
5     return int(''.join(win), 2)
```

The above algorithm first converts n into the binary representation, then adds at the end the first digit before removing it from the beginning and reconverting it into base 10.

4 | Comparisons

The algorithms from the previous Chapter were run on an M1 Macbook Air with 8GB RAM which is important because it limits the maximum recursion depth to $\sim$ 2500. Also, **NumPy** arrays were prefered over native **Python** ones, since **NumPy** is compiled natively in **C** which makes it substantially faster[1].

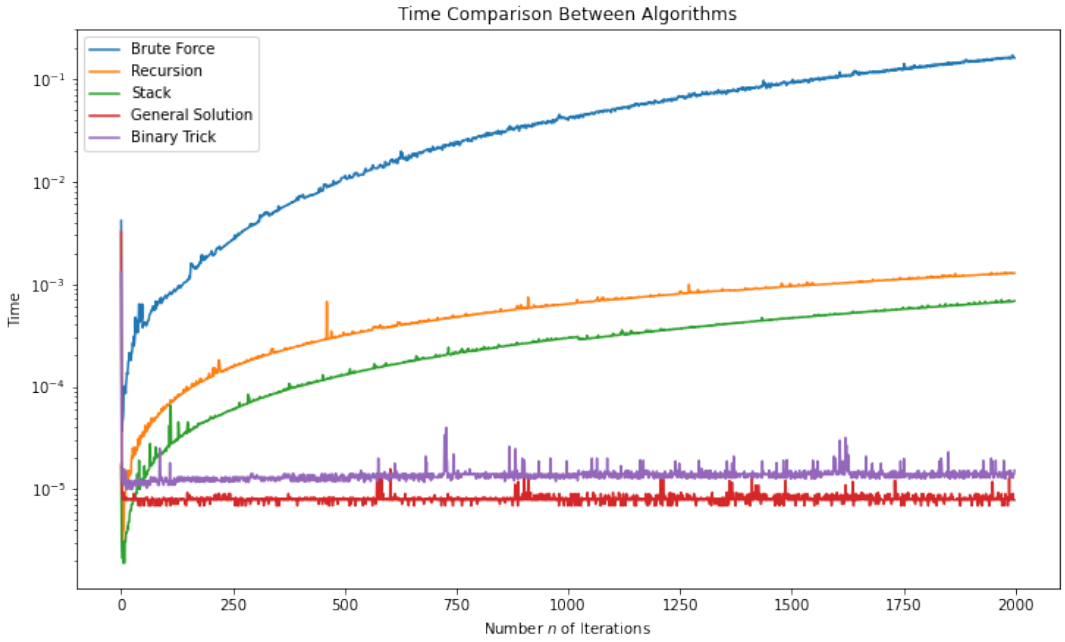

Figure 4.1: Logarithmic plot for all algorithms, where n=[1,1000].

Analyzing the above graph is simple. As expected the brute force algorithm is the slowest, and afterwards there are two pairs of algorithms, Recursion—Stack, and General Solution—Binary Trick, that appear to share the same time complexity.

Looking at the General Solution—Binary Trick in Figure 4.2 it is obvious that the prediction made in Ch. 3 regarding the complexity of the algorithms was correct and they indeed share the same complexity of $\mathcal{O}(1)$. There is a small increase as the number grows to $n = 500k$, but that can be attributed to the raise in temperature of the machine and subsequent thermal throttling, since it reached a peak temperature $T = 86^{\circ}C$ at around that point.

[1]This poses some problems with calculating the time complexity, as is discussed in the Appendix.

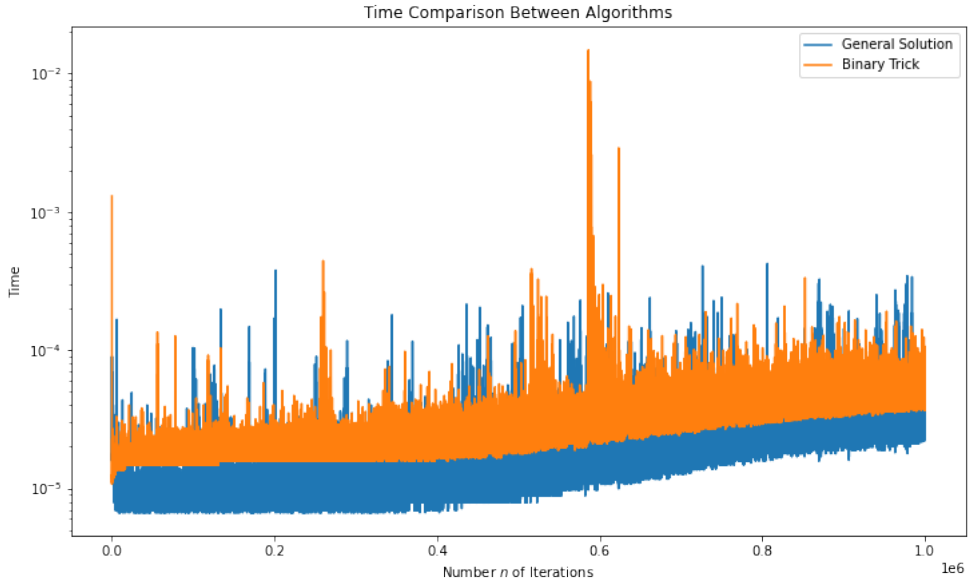

Figure 4.2: Logarithmic plot of the General Solution vs Binary Trick vs Coprime Decomposition algorithms for $n = [1, 1000000]$.

The pair Stack—Recursion is different. Unfortunately due to the limitation of the recursion depth, it is impossible to study the behaviour for big n, but from Figure 4.1 and Figure 4.3 we can see that they probably converge, but for small numbers of n they have different execution times.

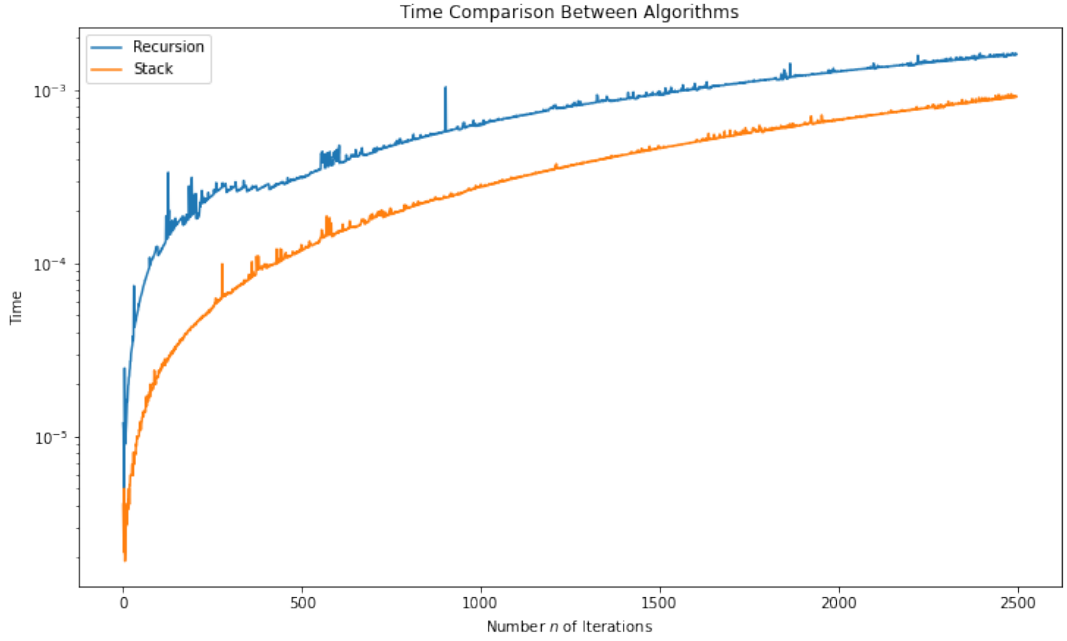

Figure 4.3: Logarithmic plot of the Stack and Recursion algorithms for $n = [1, 2500]$.

Based on the above analysis, and the fact that the only algorithms that work for varying k are the brute force, the stack, and the recursion, one can see the following

- The brute force algorithm should never be used for solving the Josephus problem, as it is way too expensive and complex. In general, brute force should be avoided, unless it is the only option.
- Using recursion is also a bad option since it's nature makes it impossible to use for big n, as it overflows the memory, and is not faster than stacking for small n.
- Using the stack method we saw the best generalizable results and having a complexity $\mathcal{O}(n)$ means that it is still fast enough for big n^2. This means that if we want to solve the Josephus problem for general k it is reliable but for $k = 2, 3$ there are better solutions, since the problem has a closed form, and the general solution algorithm is much faster.

[2]Ideally we would want $\mathcal{O}(\log n)$.

A | Derivation of Time Complexity

Here we try to explain and formalize the time complexities we noted in Chapter 2. In reality this is a rather difficult task, since **NumPy** is used alongside regular **Python**. The issue is that **Python** is an iterative language, while **NumPy** is compiled natively into **C** and is therefore orders of magnitude faster that pure **Python**. This means that the graphs in Chapter 4 should only be judged by their trajectory[1], not by the exact magnitude.

A.1 Brute Force

For the Brute Force algorithm, we have a **whileloop** what every time iterates at half the initial length, since it essentially checks the amount of people left after each cycle of killing. Then, we have a **forloop** that iterates over the entire length n of the array (each time halved by the **whileloop** and finally the last **forloop** which due to the **break** condition is difficult to calculate accurately. Never the less, as a **forloop** we can assume that in a worst case scenario it also has to iterate over the entire length n, bringing the total time complexity to $\mathcal{O}(n^3)$, although in reality the time complexity should be somewhere between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$.

A.2 Recursion

Since recursion iterates over all values of $[1, n]$, it has complexity $\mathcal{O}(n)$.

A.3 Stacking

The stack algorithm has a **forloop** iterating over all values of $[1, n]$ and a **whileloop** doing the same. However, **pop()** is of $\mathcal{O}(1)$ only for the last element and $\mathcal{O}(n)$ for an arbitrary element since the whole list has to be shifted. Normally we would expect the time complexity to be $\mathcal{O}(n^2)$, but due to the nature of **pop()**, the **whileloop–pop()** combination results in a linear complexity. So the time complexity is $\mathcal{O}(n)$.

A.4 General Solution–Binary Trick

Since for any given n we have a non increasing amount of operations, and since **NumPy** functions have stable complexity[2], specifically , this has time complexity $\mathcal{O}(1)$.

[1] In terms of time complexity.

[2] This is actually an assumption and is based on testing, since the documentation of **NumPy** provides no information.