Seattle Pacific University

Computer Science and Engineering Department

CSC 3430 Algorithm Design and Analysis

Winter 2018

On the Subject of Encryption Using the Subset Sum Problem

Grant Kalfus

Andrew Miller

Nathaniel Maciejewski

Introduction

Suppose a startup website was implementing a password login system. What is the best way to store user passwords server side? The simplest answer is in cleartext, but that carries with it several security risks. In the event of a security breach, a hacker could gain access to all user accounts, and because many users reuse passwords for multiple services, this could have further reaching consequences. Clearly some form of server-side encryption is necessary.

An optimal solution would be one where no actual password is even stored on the server itself. The server simply needs to verify the user input. On many systems, this is achieved through subsets. The server will have a large set of integers. Passwords are encrypted into subsets of these integers using an internal process (Blair, 2002). Using this method, servers simply need to store the sum of the corresponding password subset in order to verify password input. In the event of a data breach, it would be more difficult for a hacker to reverse engineer passwords simply from the set of numbers and the corresponding sums. But how safe is this system really?

Our project is on the subset sum problem. The subset sum problem is as follows: given a set of integers and a desired sum x, does there exist a non-empty subset such that the sum of all elements in the subset is equal to x? This is exactly the process that would need to be followed to reverse engineer a password from the given scenario. This project will explore the time complexity of the subset sum problem as well as implement a pseudo-polynomial time complexity solution in Python.

Background

NP stands for Nondeterministic Polynomial (Weisstien). The nondeterministic portion of the acronym refers to how it is unknown if the problem can be solved exactly in polynomial time, so the solution must be guessed through checking combinations (Eppstien, 1996). The topic of the project, the Subset Sum Problem, is NP complete (Arora, et al., 2016). This means that they are "not [polynomial] unless **P** = **NP**" (where **P** is the set of polynomial time problems) (Arora, et al., 2016). Because there is currently nothing that proves that **P** = **NP,** this illustrates the difficulty of solving the problem within a realistic timeframe. However, a solution can be reached in a reasonable amount of time with the use of a technique called dynamic programming. Dynamic programming still checks combinations, but it makes use of a map to store previously computed values to theoretically vastly speed up runtime. Our solution makes use of dynamic programming to achieve a theoretical pseudo-polynomial runtime of O(m * n), where m represents the "sum" to find, and n represents the size of a given "subset" of numbers (Kleinberg, et al., 2006).

Solution Description

The solution was implemented in Python 3.6. Conceptually, this algorithm is derivative of a video made by Roy Tushar (Tushar, 2015). To simulate large integer sets like those of a server computer, we randomly generated lists of integers of while varying list sizes to determine algorithm runtime. The set of integers was pseudo-randomly generated using the NumPy random library, bounded between 0 and *list_bounds,* a constant equal to 1000. The sum was also pseudo-randomly generated and bounded between 0 and 2 times LIST_BOUNDS, or 2000. The possible sum values were to be dynamically solved for using a 2-D array with rows corresponding to values within the integer list and columns corresponding to possible sum values. An entry set to true in this array corresponds to a value that could be summed to using a combination of integers within our set. Since the only value we are really interested in summing to is the pseudo-randomly generated sum value, called random_val in our program, the array must be only as wide as random_val to minimize runtime. Thus, our array of possible sum values, called data_keeper in our program, is n x m, where n is the length of the list of random integers and m is the desired sum random_val.

In order to dynamically determine what sums could be solved within our data_keeper array, we needed to set a few base conditions first. Column 0 of the array needed to be initialized to true. This is needed due to one of the internal conditions for checking if values can be summed to. In addition, all elements whose column number in row 0 that corresponded to a value in our set needed to be initialized to true. This is because all elements within the list can be a single element subset that sums to itself.

Figure 1: Initial Conditions for set = {1, 3, 5} with random_val = 8

	0	1	2	3	4	5	6	7	8
1									
3									
5									

Figure 1: Initial Conditions

From there, there are two checks to dynamically determine sums as the algorithm iterated through the array. The first condition: if the value one row above is true, set the current value to true. Intuitively, if one can already form a sum with earlier values, one can still form that sum while taking other values into account.

Figure 2: At row 2 (integer 3 from the set) column 1, the algorithm sets that element to true because the element above it is true.

	0	1	2	3	4	5	6	7	8
1									
3									
5									

Figure 2: Condition 1 Example

The second condition: if the value one row above and x values to the left is true (where x is the corresponding value from our set of integers indexed at the current loop value), set the current value to true. Intuitively, this means any sum that can previously formed can be added to by the current value within the set of integers to form a new sum.

Figure 3: At row 2 (integer 3 from the set) column 4, the algorithm sets that element to true because the element above it and 3 elements to the left is true.

	0	1	2	3	4	5	6	7	8
1									
3									
5									

Figure 3: Condition 2 Example

After iterating through the array, one can determine whether or not the desired sum random_val is possible by checking the final column for any true value. If one or more entries are true, the sum can be formed with a subset of values, and the program returns true.

Figure 4: All green spaces correspond to true. The algorithm has determined that 1, 3, 4, 5, 6 and 8 can all be summed to with subset {1,3,5}. It stops at 8 since 8 was the desired sum. It will iterate through column 8 until it reaches a true entry and return true. Otherwise it returns false.

	0	1	2	3	4	5	6	7	8
1									
3									
5									

Figure 4: Algorithm to Completion

Tests were run on two computers. Computer A has an Intel i7-7700K processor at 4.2GHz, with 16Gb of DDR4-2133 RAM. Computer B has an Intel i5-6600K at 3.5GHz with 16Gb of DDR4-2133 RAM.

To determine the runtime of our algorithm, we varied both n (the length of the set or integers) and m (the sum being searched for) separately. We varied n from 1 to 1000. We varied m from 1 to 1000 and scaled it by a pseudo-randomly generated integer bounded from 0 to 10.

Results

Computer A

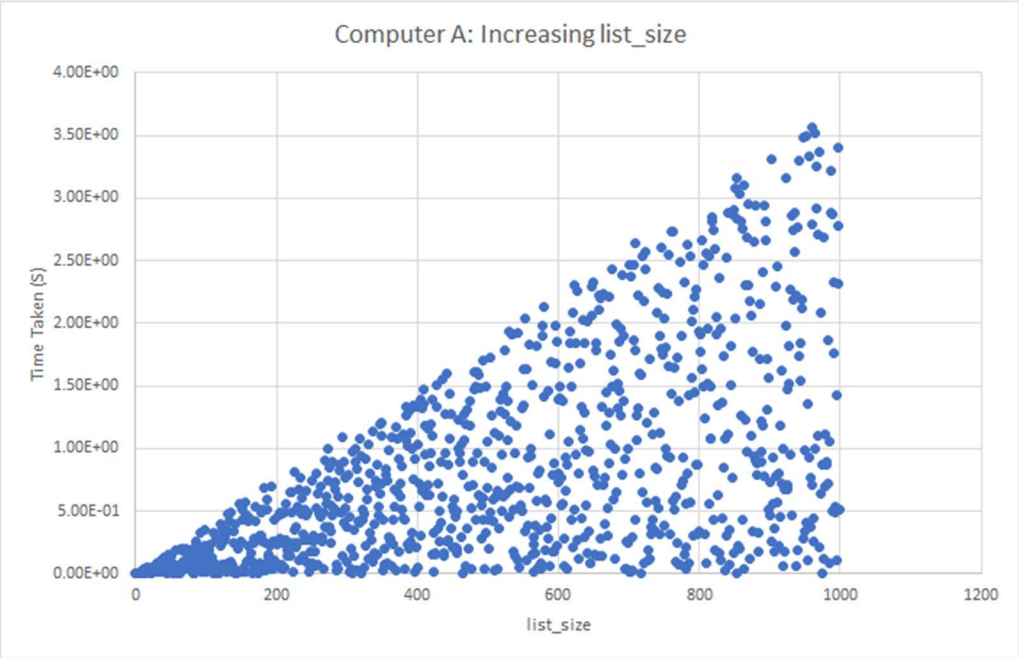

Figure 5. Increasing list_size for computer A

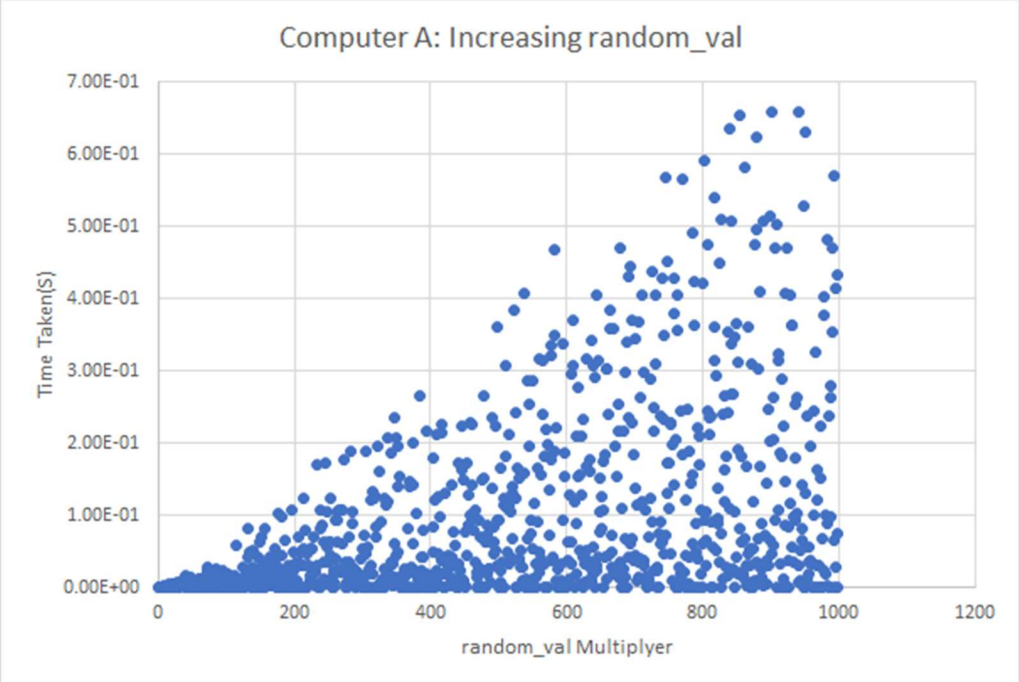

Figure 6. Increasing random_val for computer A

Computer B

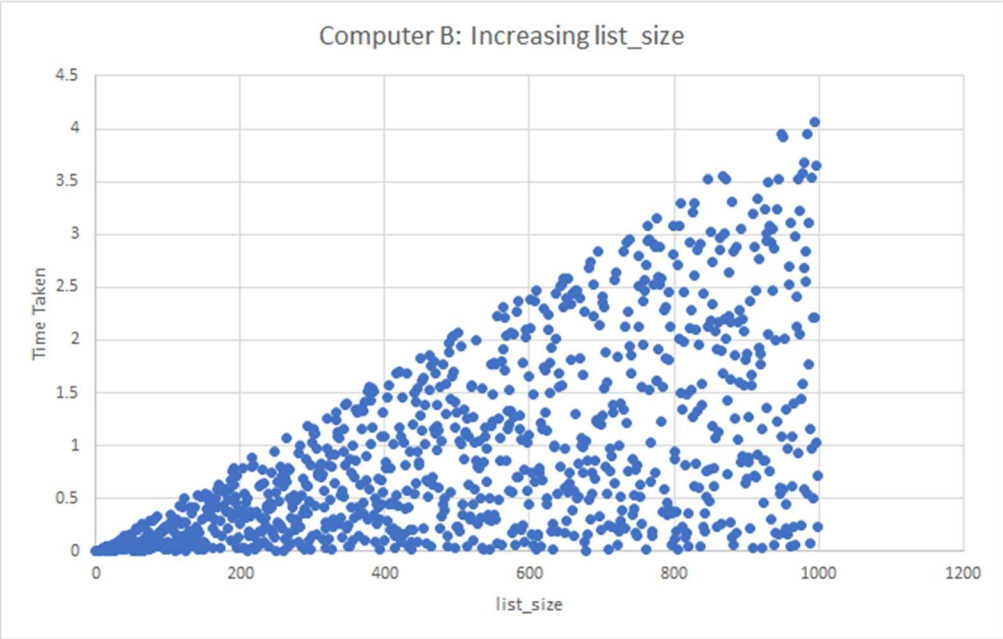

Figure 7. Increasing list_size for computer B

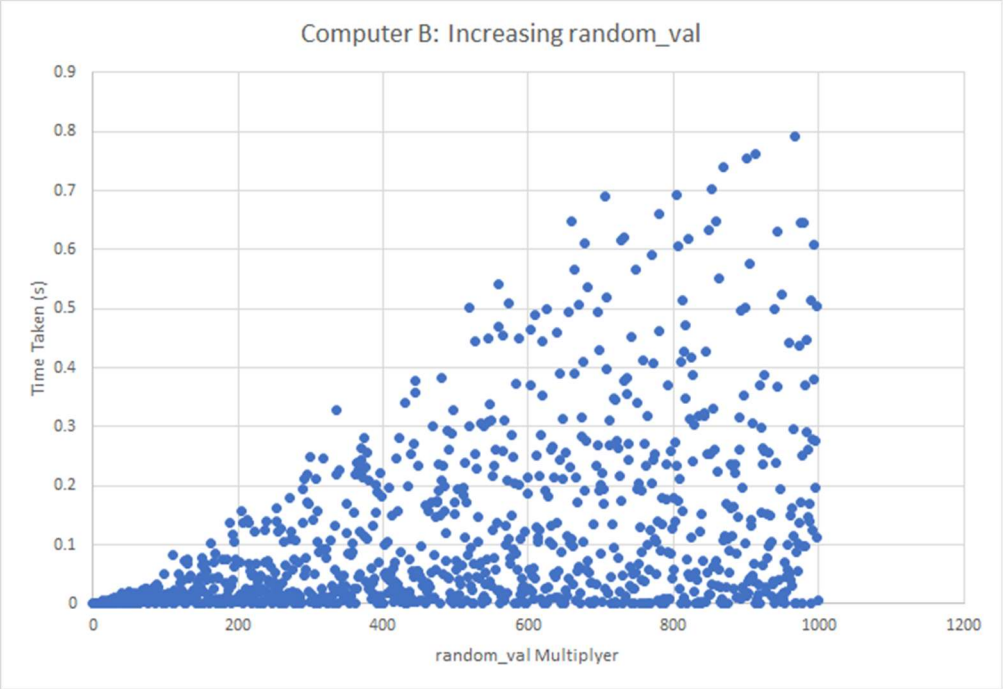

Figure 8. Increasing random_val for computer B

As can be seen from Figures 5-8, using a static list size and a variable random target value multiplier causes performance to vary linearly with the multiplier, and using a static multiplier with a variable list size causes performance to vary linearly with the list size. Taken together, this means we have a complexity class of max_list_size*max_target_value, which corresponds to the pseudo-polynomial runtime of O (n * m) we expect to see from the dynamic programming algorithm (Kleinberg, et al., 2006).

Conclusion

Ultimately, the runtime of our algorithm being pseudo-polynomial demonstrates that subset encryption is not a wholly safe encryption technique, as it is not as difficult to reverse engineer subsets corresponding to passwords as it initially appears. However, to fully reverse engineer a password knowledge of the process of encryption between a password string and the subset is required so that the password string can be reverse engineered from the subset itself. Furthermore, the algorithm can be slowed down substantially with large integer sets and high subset sum values. If implemented properly, there is still potential for a safe encryption method using subsets (Blair, 2002).

Our learning experience was enhanced by a false start: we initially planned to work with 3SAT and circuit-solver problems, eventually changing our minds and moving to a more easily-understood problem. From studying two NP problems, especially since 3SAT is important for our proof that SSP is NP-complete, we ultimately gained a better understanding of how NP problems are related and NP class functions as a whole (including NP, NP-complete and NP-hard) that we may not have otherwise. We also got a useful chance to practice directly with coding dynamic programming algorithms.

All of this matters because lessons learned through analyzing past errors tend to stick better than those imparted first try; as such, this understanding of the NP class as a whole is much more likely to be retained and usable later on in the professional world.

A more practical asset we gained from this project was a better appreciation for dynamic programming. As all problems in NP can be reduced to any NP-complete problem, any problem in NP can be solved through dynamic programming, provided we recognize it and are able to correctly analyze and set it up. For a class of problems notoriously difficult and time-consuming to produce the optimal answer for, this is extremely valuable and a useful part of a real-world skillset.

References

Arora, Sanjeev and Barak, Boaz. 2016. *Computational Complexity: A Modern Approach.* New York : Cambridge University Press, 2016. 0521424267.

Blair, Charles. 2002. Other Uses of the Sumset Sum Problem. *Stony Brook University Mathmatics Department.* [Online] Institute for Mathematical Sciences, Stony Brook University, 12 14, 2002. [Cited: 03 04, 2018.] http://www.math.stonybrook.edu/~scott/blair/Other_uses_subset_sum.html.

Eppstien, David. 1996. *University California Irvine.* [Online] March 12, 1996. [Cited: March 4, 2018.] https://www.ics.uci.edu/~eppstein/161/960312.html.

Kleinberg, Jon and Tardos, Éva. 2006. *Algorithm Design.* s.l. : Pearson Addison Wesley, 2006. 0-321-29535-8.

Tushar, Roy. 2015. *Subset Sum Problem Dynamic Programming.* Youtube, 2015.

Weisstien, Eric. NP-Problem. *Wolfram Alpha.* [Online] [Cited: March 4, 2018.] http://mathworld.wolfram.com/NP-Problem.html.