View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digital Commons @ Butler University

Butler University

Digital Commons @ Butler University

Undergraduate Honors Thesis Collection Undergraduate Honors Thesis Collection

2020

The Knapsack Subproblem of the Algorithm to Compute the Erdos-Selfridge Function

Brianna Sorenson
Butler University

Follow this and additional works at: https://digitalcommons.butler.edu/ugtheses

Part of the Computer Sciences Commons

Recommended Citation

Sorenson, Brianna, "The Knapsack Subproblem of the Algorithm to Compute the Erdos-Selfridge Function" (2020). *Undergraduate Honors Thesis Collection*. 513.
https://digitalcommons.butler.edu/ugtheses/513

This Thesis is brought to you for free and open access by the Undergraduate Honors Thesis Collection at Digital Commons @ Butler University. It has been accepted for inclusion in Undergraduate Honors Thesis Collection by an authorized administrator of Digital Commons @ Butler University. For more information, please contact digitalscholarship@butler.edu.

The Knapsack Subproblem of the Algorithm to Compute the Erdős-Selfridge Function

A Thesis

Presented to the Department of Computer Science and Software Engineering

College of Liberal Arts and Sciences

and

The Honors Program

of

Butler University

In Partial Fulfillment

of the Requirements for Graduation Honors

Brianna Sorenson

May 10, 2020

Contents

1 Introduction

After explaining the g(k) function and how to compute it, we will explore ways of solving the knapsack subproblem embedded in this algorithm.

1.1 The g(k) Problem

The Erdős-Selfridge Function, which I will continue referring to as g, takes an integer input k and returns the smallest integer $n > k+1$ such that the smallest prime divisor of the binomial coefficient $\binom{n}{k}$ is greater than k. To get a better sense of the problem, we can use Pascal's triangle [6] to get the first few values explicitly:

	0	1	2	3	4	5	6	7
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

A simple way to find $g(k)$ for small k is to check each integer in column k, starting at row $k+1$, until you find the first integer whose smallest prime factor is greater than k, and the answer will be that row number. For $k = 2$ we start looking in column 2 (1, 3, 6, 10, 15, 21, ...) and start with 6 because it is on row 4. It is divisible by 2, so we move on to 10, which is also divisible by 2. We move on to 15, whose smallest prime factor is 3, so our answer for $g(2)$ will be the number of the row we found 15 in, so

$$g(2) = 6.$$

Similarly, we find that

$$g(3) = g(4) = 7$$

because 35's smallest prime factor is 5, and for both k's, it is the first number in their column which is not divisible by 2 or 3.

This is not the method we use in the algorithm, as the values of $g(k)$ grow very quickly, and generating Pascal's triangle for well over 100 rows is a lot of work we do not need to do. Instead, we use a kind of sieve called a wheel which works using the following theorem.

Theorem 1.1 (Kummer's Theorem) *[3]*

Let $k < n$ be positive integers, and let p be a prime $\leq k$. Let t be a positive integer with $t \geq \log_k n$. Write

$$k = \sum_{i=0}^{t} a_i p^i \quad \textit{and} \quad n = \sum_{i=0}^{t} b_i p^i$$

as the base-p representations of k and n respectively. Then p does not divide $\binom{n}{k}$ if and only if $b_i \geq a_i$ for $i = 0, \ldots, t$.

As an example, we will use Kummer's Theorem to see if $\binom{7}{4}$ is divisible by 3. First, we will write 7 in base 3: 21_3; and we will write 4 in base 3: 11_3. the base-3-digits (or trits) of 21_3 are all greater than or equal to 11_3, so $\binom{7}{4}$ does not have a factor of 3. And we know this from our earlier exercise with Pascal's triangle. Next, we will see if $\binom{7}{5}$ is divisible by 3. We recall 7 base 3 is 21_3, and now we see that 5 base 3 is 12_3. The last trit of 12_3 is greater than the last trit of 21_3, so $\binom{7}{5}$ must have a factor of 3. From Pascal's triangle, we can see that $\binom{7}{5} = 21$ which is clearly a multiple of 3.

The main idea behind how the algorithm we use to compute the Erdős-Selfridge function works is that we need to use the information that Kummer's theorem gives us about the divisibility of $g(k)$ for each prime power and the Chinese remainder theorem will allow us to combine those separate moduli into one, so that we can get an answer that is admissible for each one.

Theorem 1.2 (Chinese Remainder Theorem) *Let p, q be coprime. Then the system of equations*

$$x = a \mod p$$

$$x = b \mod q$$

has a unique solution for x modulo pq.

We use a wheel sieve to achieve this aim. A wheel is a data structure made up of rings, which each have a modulus and a set of remainders for that modulus, and it enumerates the solutions to the Chinese remainder theorem quickly. As an example, if you wanted to get numbers that are 3 or 5 mod 7 and 9 or 10 mod 11, the wheel would have two rings: one for the desired remainders mod 7, and one for the desired remainders mod 11. The wheel made with these rings would then spit out the numbers 10, 31, 54, and 75. Later on we will go through an example of how this works. In our application of a wheel sieve for the Erdős-Selfridge function, we only use a subset of the rings we construct in the main sieve itself, and once we get through the explanation of why using a wheel for this problem is important, we will discuss how and why we exclude some of our rings. [1, 2]

Here is an example of how we use Kummer's theorem to make a ring for $k = 10$. For each prime $p \leq 10$, we write k in base p. So, 10 would be 1010_2 in base 2. Therefore, we will admit the remainders 1010_2, 1110_2, 1011_2, and 1111_2. Converting this back into decimal, we get 10, 11, 14, and 15 mod 16.

We then use the same method to find this information for the other primes smaller than 10 so that we may use them as rings for our sieving algorithm. So, using our $k = 10$ example, the ring for $p = 2$ would have an array of length 16 where every entry is a zero except the spots for the admissible remainders, so the 10th, 11th, 14th, and 15th spots will each have a 1, like this:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1

Now, in order to quickly find the numbers we want from this ring, we are going to include another array of the same length that tells us how much we need to add to any arbitrary number in order to get a number with the next admissible remainder mod 16. Because remainders cycle around like a clock, the numbers for this array will too. It will look like this:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+10	+9	+8	+7	+6	+5	+4	+3	+2	+1	+1	+3	+2	+1	+1	+11

The reason we do this is because incrementing by one and checking the remainder over and over until we get to a number we want takes more time than constructing this second array once and knowing we will be at a correct point once we add the number in the right cell of the array. So with an example, we start at 0, and the jump array tells us to add 10 in order to get to a number with an admissible remainder mod 16. Now we have the number 10. We will then be able to send this number to the next ring, and because this ring has 4 admissible remainders we will continue to send 3 more numbers. Our number 10 is 10 mod 16, so the jump array tells us to add 1 to get 11. We will send 11 to the next ring, and then add the next jump value: 3. We send 14 to the next ring, add 1, then send 15. We have now sent all 4, and the work of this ring is done. In this example it seems like we could have just sent the remainders we got initially from using Kummer's theorem, but it becomes more apparent why these jump arrays are important when we see what happens when these numbers get to the next ring.

Here, we can do an example where we fully compute the value of $g(5)$. I have the rings we will be using with their admissible remainders and jump arrays constructed below:

$k = 5$	0	1	2	3	4	5	6	7	8
8	0	0	0	0	0	1	0	1	
	+5	+4	+3	+2	+1	+2	+1	+6	
9	0	0	0	0	0	1	0	0	1
	+8	+16	+24	+32	+40	+48	+8	+16	+24

As you can see, the jump values of the 9 ring are all multiples of 8. This is important because when we are adding to numbers in the 9 ring that we got from the 8 ring, we do not want to change the number's remainder mod 8 because we know that it is already admissible for that ring. Because there are two admissible remainders in each ring, we will get 4 numbers from this wheel. First, we start with 0. It is not admissible to the 8 ring, so we will add 5. now we have 5, which is admissible, so we will send it to the 9 ring. In the 9 ring, 5 is an admissible remainder, so we have 5 as one of our four final numbers. Still in the 9 ring, we will add 48 to get the next number in our final four: 53. Now, we will go back to the 8 ring. We are still holding on to that 5 we sent the 9 ring earlier so that we can now add 2, and send a 7 to the 9 ring. 7 is not admissible for the 9 ring, so we will add 16 and send 23 to the final 4. 23 is 5 mod 9 so we will add 48 and send 71 as our last number in the final set. Now we are left with 5, 53, 23, and 71. One of these numbers is $g(5)$. You may have noticed that we omitted the ring constructed for the prime 5. While we did not use it in the wheel, we can still use it to eliminate some of our options here. The ring would have admitted all remainders ≥ 5 mod 25, so we can eliminate 53. We also know that $g(5) \geq k+1$ so we can eliminate 5. Now we are left with 23 and 71. Of these, we want the smallest answer, so

$$g(5) = 23.$$

As you can likely see at this point, for larger and larger k there will be more and more rings and therefore the work we do will grow more and more. However, in our example to find $g(5)$, we excluded the ring for 5 and still got the correct answer because the product of our included ring lengths is larger than our final answer. We actually only need the product of the sizes of the rings to be *just* larger than our final answer. The closer it is, the smaller our range of numbers the wheel will give us. However, we also want our rings that we do include to have as few admissible remainders as possible because the number of candidate answers the wheel gives us is the product of the number of admissible remainders in each ring. Therefore, we could do less work to find the solution for $g(k)$, but we need some method of determining which rings we want to keep and which rings we want to exclude from our sieve.

1.2 The Knapsack Problem

There is a similar problem of choosing between things to keep and things to exclude known as the knapsack problem. Imagine you have broken into a fancy art museum and you only have a certain amount of time to take what you want, so you can only make one trip and leave with only the things you can carry in the knapsack you brought with you. Your goal is to grab things you can sell to a fence and get as much money as possible. Therefore, you will probably want to prioritize grabbing small and expensive things like ancient jewelry found from important archaeological sites. Even though that huge modern art sculpture would get you enough money that you'd never have to commit crime in metaphorical thought experiment art museums ever again, there is simply no way you will ever be able to fit it into your knapsack. So, you could use a knapsack algorithm to find the best combination of items to take with you so that you get the maximum possible value for your available capacity.

1.3 Thesis Statement

We can use one of the existing knapsack algorithms to select which rings we want to use in our wheel. There are 3 popular algorithms to solve this problem: the greedy approach, the branch-and-bound approach, and the dynamic approach. We used all three of these methods to select combinations of rings so that we can compare the time it takes each of them to complete the task and see which of them gives us the best answers. We suspect that the greedy algorithm will prove to be the best choice here for two reasons: it works well with split rings, which will be explored in more detail soon enough, and the fact that we have a large number of items with similar mass to select from.

1.4 How We Make the Knapsack Algorithms Work For Us

The unmodified knapsack algorithms work additively, which makes sense in the art heist context because if you take a vase worth 40,000 dollars weighing 30 pounds and a painting worth 35,000 dollars weighing 15 pounds, you have 75,000 dollars worth of things in your knapsack weighing a total of 45 pounds. However, for our problem, if you include a ring of size 16 with 4 admissible remainders and a ring of size 27 with 12 admissible remainders, you end up with a 'mass' of 432 and a 'value' of 48 admissible remainders overall, so in order to convert our multiplicative value and mass into an additive algorithm, we simply take logarithms.[1]

We actually want to minimize the number of admissible residues while knapsack algorithms maximize value, so our equation for the value of a ring will need to be larger for fewer admissible remainders. This is our formula for the value of a ring:

$$\log\left(\frac{\text{size of ring}}{\text{number of admissible remainders}}\right)$$

[1]Recall that $\log(ab) = \log(a) + \log(b)$.

And we calculate the mass of each ring with this formula:

$$\log(\text{size of ring})$$

We calculate the size of the knapsack as:

$$\log(\hat{g}(k)) + \log(k)$$

Here, $\hat{g}(k)$ is an estimate of $g(k)$ and we include a fudge factor of k to ensure the knapsack is large enough to contain $g(k)$ in the candidate solution set for the wheel. [8]

Right now, we are using a greedy algorithm to select which rings we want to use in our main calculations, which is fast but not necessarily optimal. It works by sorting the rings by the ratio of value to mass and takes the ones with the best ratios in order until the knapsack is too full to fit more items. This method has so far been successful, as we have now used it to calculate values for k well over 300. Previous algorithms were unable to compute $g(200)$ in less than a month while using specialized hardware and multiple processors, but our algorithm has been able to calculate this same value in around an hour using only a single general purpose processor. [7, 8]

While our current method works well enough, my aim is to see if it would be worth the extra time to use a knapsack algorithm which is optimal for other applications of the knapsack problem, and if so, discover what implementation thereof would be best to use for our case.

Splitting Prime Rings

Our application is actually a slight variation of the original knapsack problem because we have a method to split up prime rings into smaller rings which may retain most of the filtering value of those rings while taking up less weight. This difference will require us to make changes to the established optimal solutions to account for this, since splitting the rings will be made redundant if the solutions given by the knapsack algorithms include more than one split from the same prime ring, so we must find a way to exclude such cases.

For example, using our length 16 ring for $k = 10$, we could make smaller rings using what we know from our original, so long as they all have lengths that are powers of p, which in this case is 2.

16	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1
8	0	0	1	1	0	0	1	1								
4	0	0	1	1												
2	1	1														

As you can see here, since 10, 11, 14, and 15 are 2, 3, 6, and 7 mod 8 respectively, we have a smaller ring which will admit all the same numbers we want it to admit but it will also admit numbers which we want to eliminate. This would normally be a problem, but as long as we keep the 16 length ring for filtering at the end, we still eliminate the candidate answers we do not want. However, splitting rings creates new problems which require attention. Each smaller ring from the same prime will never have greater value than the larger ones. In this example, we can see that the length 8 ring is less valuable than the length 16 ring because they have the same number of admissible residues. Also, the length 2 ring is completely worthless.

This also shows why it is redundant to include any two of these splits in our wheel. Because bigger rings always have greater or equal filtering value than smaller ones, including more than one from the same prime does not give us any new information. In this particular case, you can see that it is likely better to just use the biggest ring, but sometimes the rings we get end up being like the 8 length one is here, where the length 4 ring has exactly the same value with a smaller size, and so choosing 4 over 8 would clearly be the correct choice. This illustrates why having a specialized knapsack algorithm which takes this into account is so beneficial.

2 Knapsack Background

2.1 $\mathcal{NP}$-Complete

The knapsack problem is part of a class of problems known as $\mathcal{NP}$-complete. A problem is in $\mathcal{NP}$ if a candidate solution to that problem can be verified in polynomial time, which means that the time it takes for the function to run is some polynomial function of the input. A problem is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ *and* every problem in $\mathcal{NP}$ is reducible to that problem in polynomial time. This is interesting because there are also many other problems that are $\mathcal{NP}$-complete, like the travelling salesperson problem, the graph coloring problem, and the Hamiltonian path problem to name only a few. The best known algorithms for these types of problems are exponential in time, and for a problem to be considered feasible, it must have a solution which runs in polynomial time. This essentially means that $\mathcal{NP}$-complete problems are quite hard to solve quickly. [5, 4]

2.2 Greedy, Branch-and-bound, and Dynamic Algorithms

The greedy algorithm works by sorting the items according to some criteria and takes the items in order from the top of the list until there is no more room in the knapsack to hold more items. Some common criteria used to determine the sort include most to least valuable items, least to most massive items, and greatest to least ratio of value to mass. This last criteria is the one we selected for our algorithm, and it has been working well for us so far. The greedy algorithm, although just as fast as whichever sorting algorithm is chosen for the items, cannot guarantee an optimal solution, unlike the two of the common optimal knapsack algorithms which I will explain later. However, with our added feature of ring splitting, these so-called optimal solutions may not be as effective as they are for the general problem, so this is still anyone's game. For more information on knapsack algorithms, see the book *Knapsack Problems* by Kellerer, Pferschy, and Pisinger. [5]

2.3 Why These?

Greedy, as has been said before, is the simple quick and dirty method which gives us good enough solutions to be very helpful indeed to this problem. It is the easiest, laziest option. The dynamic approach is slower and optimal for the usual knapsack cases but it is difficult to adapt to our splitting criteria. Branch-and-bound is slower still but optimal for the general problem and far more easily customized to our specifications than dynamic. These three options give us a good amount of variety for our purposes so we have room to play around with them.

3 Experimental Results

3.1 Greedy vs. Optimal Solutions

The greedy algorithm is by far the most simple algorithm for the knapsack problem. All it does is sort the items and then it takes those items from the top of that newly sorted list until the next item cannot fit into the knapsack. Therefore, it is just as fast as whatever sorting algorithm you chose to use, which in our case is $O(i^2)$ because we are using selection sort, but there are algorithms which are asymptotically faster, like quick sort or merge sort, which are $O(i \log(i))$. We chose to use selection sort with the idea that it minimizes the number of data moves, which is an expensive procedure time-wise because we are moving ring data structures around. This algorithm does not guarantee an optimal solution to our problem, though. The other algorithms we are using are optimal for the general knapsack problem and may have a chance at retaining that property for our specific application with a few tweaks.

3.2 Branch-and-bound

The branch-and-bound method of solving the knapsack problem has time complexity $O(2^i)$, with i being the number of items to consider, which is quite slow at first glance, but this is a worst-case estimate. In reality, it is a bit faster when adding in a few clever tricks. Essentially, the algorithm works by exploring every possible combination of items in a depth-first manner. It will consider all combinations which include the first item, the first and second item, ..., combinations which do not include the first item and do include the second, etc. However, we can make this algorithm faster by telling it to abandon branches of its decision tree by using a heuristic. So long as we keep track of the greatest value of a combination we have seen so far, and have an estimate of the maximum value we could get from any combination further down in the next branch of our decision tree, when the heuristic tells us that the remaining combinations in the rest of the branch will not be able to add up to a value as high as a combination we have already seen, we can prune off the rest of the branch by skipping it entirely. This can save us a lot of work, especially if it prunes off a branch which is close to our root node.

This algorithm is very good for dealing with altered versions of the knapsack problem, like what we have with our splits, because it is a simple tweak to code in conditions for where the tree should prune more branches. If the knapsack includes two rings from the same prime, abandon this branch. This allows us to make the algorithm avoid combinations we want to avoid. This algorithm is also useful in that it can work unaltered with non-integer values for capacity of the knapsack and size of the items, which cannot be said of the dynamic algorithm. This is important to us because we are using non-integer values for both our mass and value variables.

3.3 Dynamic Programming

If branch-and-bound works using a decision tree, dynamic works using a grid. The algorithm first constructs a two-dimensional array with the dimensions being 1 plus the capacity, W, of the knapsack and 1 plus the number, i, of items. Then, beginning in the corner which represents 0 items and 0 capacity of the knapsack, the algorithm fills in that place with the highest value which can be made with that subset of items and that capacity. By working from the bottom in this manner, the calculations for each individual cell of the array are very fast and simple, requiring only constant time. In this example, we have the number of items considered as the vertical axis and the capacity allowed as the horizontal axis, with the items in the table on the right:

	0	1	2	3	value	weight
0	0	0	0	0	10	1
1	0	10	10	10	20	3
2	0	10	10	20	30	2
3	0	10	30	40		

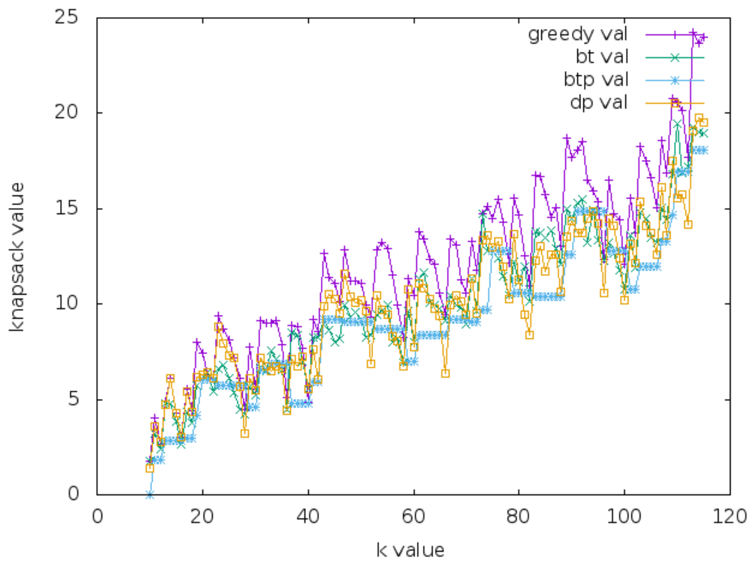
Eventually, the array builds up to the very last cell in the very last row and column of the array, which contains the optimal possible value. Therefore, it makes sense that the time it takes would simply be equal to the number of cells in this array: $O(iW)$.[2]

This algorithm can be much faster than branch-and-bound when working with a small enough knapsack capacity, but because arrays must have integer dimensions, and we are working with non-integer values for both capacity of the knapsack and size of the items, we must accept the necessary evil of rounding errors. Also, due to the nature of how the values of item combinations are calculated, we cannot prune away unhelpful or unacceptable combinations for our special splitting condition as we could with a decision tree. It is likely that there is a way to achieve a similar effect, but implementing such a thing would add to its run time rather than lessen it. It is therefore more difficult to make it come up with the better solutions that splitting provides and would lead to the greedy algorithm finding better solutions, as we have discovered.

3.4 Computations

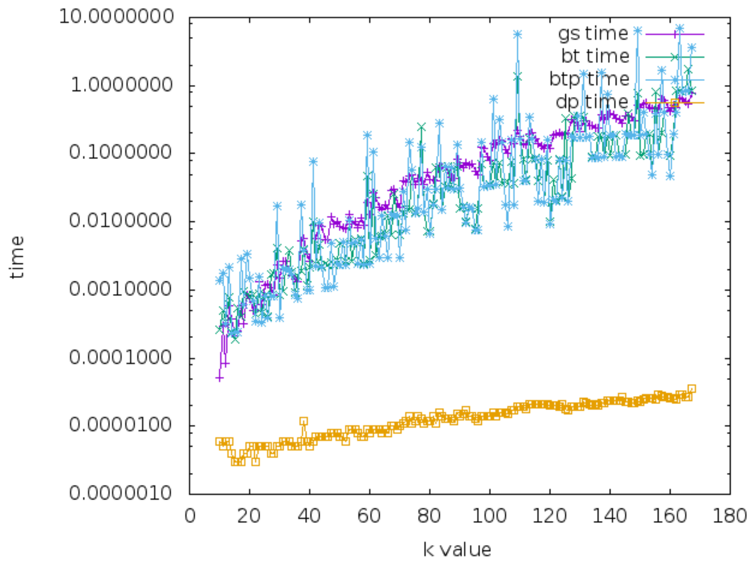
We have data comparing our current greedy algorithm which uses split rings (greedy), a branch-and-bound backtracking algorithm using splits which excludes combinations that include two rings from the same prime (btp), another branch-and-bound which doesn't bother to prune away combinations with rings from the same prime but instead unsplits any pairs which make their way into the solution by simply removing the smaller one (bt), and a normal dynamic algorithm which doesn't bother with splitting (dp).

[2]This may seem like polynomial time, but in actuality W could, for the general problem, be anything at all. It may be i^2, or 4563827^i, or 13. In our case, it's around $\frac{k}{\log(k)}$, if we round all masses to the nearest integer, giving up on optimality.



Strangely, although greedy is not an optimal algorithm, it is a consistent contender for greatest value found while the other three methods seem to bounce around in their results. Greedy does not go undefeated, but especially with the larger values of k, the difference is apparent. It seems that the splitting condition was enough to revoke the title of 'optimal' from these other algorithms.

Here is a graph of the time it took for each one.



It is clear that dynamic takes the win here for time. Greedy can likely be improved by using a different sorting algorithm, and it is more consistent than the branch-and-bound algorithms. The large spikes from the branch-and-bound algorithms, especially the pruning version, show them to be poorer than greedy time-wise.

4 Conclusions and Future Work

It seems that our initial assumption that the greedy algorithm would do better than the alternatives has been proven correct. I would have expected the other algorithms to perform at least as well as the greedy algorithm in terms of maximum value combination found and that the sticking point would have been in how slow they are to execute, but the greedy algorithm is clearly the best choice. This is likely because all our rings have similar sizes overall. To bring back the thief analogy, imagine you are robbing a jewelry store, but the only things they sell are rings. All these items have similar weight, and your knapsack is quite small. There is a wider range of how valuable each ring is though, so just grabbing a random handful could leave you with a pretty poor haul. The dynamic and branch-and-bound approaches both take the weight of each item as an important variable where that is not as necessary for our specific case. The greedy method of just taking the rings with the most valuable gems has proven to be the most effective approach.

The greedy algorithm also has the easiest method of dealing with split rings. If two rings split from the same prime have a great enough value/mass ratio that both would be included for the wheel, the smaller of the pair of rings can simply be deleted from the sorted list of rings. This process is repeated until the solution no longer contains more than one ring split from the same prime.

5 Code

```
#include <iostream>

#include <fstream>

#include "smallprimes.h"

#include "ring.h"

#include "estimate.h"

#include <time.h>


clock_t start;

clock_t ending;

double bt_time_used, btp_time_used, dp_time_used, g_time_used = 0;


// sorts rings in this array by value/size


void sortbyratio(vector<Ring> &r)

{

        for (int i = 0; i + 1 < r.size(); i++)

        {

                for (int j = i + 1; j < r.size(); j++)

                        if (r[j].ratio > r[i].ratio)

                        {

                                Ring x = r[j];

                                r[j] = r[i];

                                r[i] = x;

                        }

        }

}


// returns the total value of all the rings in this array


double val(vector<Ring> r)

{
        double toreturn = 0;

        for (int i = 0; i < r.size(); i++)

        {

                toreturn += r[i].value;

        }

        return toreturn;

}


// returns the total size of all the rings in this array


double siz(vector<Ring> r)

{

        double toreturn = 0;

        for (int i = 0; i < r.size(); i++)

                toreturn += r[i].size;

        return toreturn;

}


vector<Ring> btknapinc, btpknapinc;

double btknapval, btpknapval = 0;

double btknapsizeleft, btpknapsizeleft = 0;


// recursive part of btpknap()


void btpknaprec(double possval, vector<Ring> total,

        vector<Ring> included, int place, double value,

        double capacity)

{

        if (capacity < total[place].size) return;   // if there's no room, stop


        if (value >= btknapval) // if best value so far, save it

        {

                btpknapval = value;
                btpknapsizeleft = capacity;
                btpknapinc = included;
        }

        if (possval < btknapval) return;
        // if we cannot get a better value on this branch, stop

        if (place >= total.size()) return;  // if no items left, stop
        for (int i = 0; i < included.size(); i++)
        {   // if we have two rings from the same prime, stop
                if (total[place].p == included[i].p)
                {
                        vector<Ring> without = included;
                        btpknaprec(possval - total[place].value, total,
                                without, place + 1, value, capacity);
                        return;
                }
        }

        vector<Ring> with = included;
        with.push_back(total[place]);
        btpknaprec(possval - total[place].value, total,
                with, place + 1, value + total[place].value,
                capacity - total[place].size);  // include item
        vector<Ring> without = included;
        btpknaprec(possval - total[place].value, total,
                without, place + 1, value, capacity);   // don't include item
}

// the branch-and-bound algorithm which prunes splits

void btpknap(vector<Ring> r, double W)
{
        start = clock();    // start timer
        vector<Ring> included;
        included.resize(0);
        btpknaprec(val(r), r, included, 0, 0.0, W); // real work is here
        ending = clock();   // end timer
        btp_time_used =
                ((double)(ending - start)) / CLOCKS_PER_SEC;
}

// recursive part of btknap()

void btknaprec(double possval, vector<Ring> total,
        vector<Ring> included, int place, double value,
        double capacity)
{
        if (capacity < 0) return;   // if there's no room, stop
        if (value >= btknapval) // if this is the greatest value so far, save it
        {
                btknapval = value;
                btknapsizeleft = capacity;
                btknapinc = included;
        }
        if (place >= total.size())  // if theres no more items, stop
        {
                return;
        }
        if (possval < btknapval) return;
        // if we cannot do better than the maximum so far, stop
        vector<Ring> wonewitem = included;
        wonewitem.push_back(total[place]);
        btknaprec(possval - total[place].value, total,
                wonewitem, place + 1, value + total[place].value,
                capacity - total[place].size);  // include this item
        vector<Ring> wnewitem = included;
        btknaprec(possval - total[place].value, total,
                wnewitem, place + 1, value, capacity);  // don't include this item
}

// the branch-and-bound algorithm without pruning splits

void btknap(vector<Ring> r, double W)
{
        start = clock();    // start timer
        vector<Ring> included;
        included.resize(0);
        btknapinc.resize(0);
        btknapval = 0;
        btknapsizeleft = 0;
        btknaprec(val(r), r, included, 0, 0.0, W);  // calls function
        ending = clock();   // end timer
        bt_time_used =
                ((double)(ending - start)) / CLOCKS_PER_SEC;
}

vector<Ring> dpknapinc, gsplitknapinc, gknapinc;
double dpknapval, gsplitknapval, gknapval = 0;
double dpknapsizeleft, gsplitknapsizeleft,
gknapsizeleft = 0;

// the dynamic programming knapsack algorithm

void dpknap(vector<Ring> r, int W)
{
        start = clock();    // start timer
        dpknapinc.resize(0);
        double **mat;
```

```
mat = new double *[r.size() + 1];
// construct 2d array that is number of items by knapsack size
for (int i = 0; i <= r.size(); i++) // initialize the 0 values
        mat[i] = new double[W + 1];
for (int i = 0; i < r.size() + 1; i++)
        mat[i][0] = 0;
for (int i = 0; i < W + 1; i++)
        mat[0][i] = 0;
for (int item = 1; item <= r.size(); item++)
{
        for (int capacity = 1; capacity <= W; capacity++)
        {
                double maxValWithoutCurr =
                        mat[item - 1][capacity];
                double maxValWithCurr = 0.0;
                int weightOfCurr =
                        r[item - 1].size + .5;
                if (capacity >= weightOfCurr)
                {   // if there's room for this item, add it as new max value
                        maxValWithCurr =
                                r[item - 1].value;
                        int remainingCapacity =
                                capacity - weightOfCurr;
                        maxValWithCurr +=
                                mat[item - 1][remainingCapacity];
                }   // put the maximum value in this spot of the array
                mat[item][capacity] =
                        (maxValWithoutCurr < maxValWithCurr) ?
                        maxValWithCurr : maxValWithoutCurr;
        }
}
dpknapval = mat[r.size()][W];
for (int i = 0; i <= r.size(); i++)
```

```
                delete mat[i];
        delete[] mat;
        ending = clock();   // end timer
        dp_time_used =
                ((double)(ending - start)) / CLOCKS_PER_SEC;
}

// a function to print all the data for a particular k value to one file

void print(int k, const vector<Ring> &r, int cutoff)
{
        char filename[] = "xxx.txt";
        filename[0] = '0' + k / 100;
        filename[1] = '0' + (k / 10) % 10;
        filename[2] = '0' + k % 10;
        ofstream file(filename);

        file << "greedy split" << endl;
        for (int i = 0; i < gsplitknapinc.size(); i++)
        {
                file << gsplitknapinc[i].p << " ";
        }
        file << gsplitknapsizeleft << endl;
        file << gsplitknapval << " " << g_time_used << endl;

        file
                <<
                "bt with branch and bound and splits then unsplit after" <<
                endl;
        for (int i = 0; i < btknapinc.size(); i++)
        {
                file << btknapinc[i].p << " ";
        }
        file << btknapsizeleft << endl;
        file << btknapval << " " << bt_time_used << endl;

        file << "bt with pruning and splits" << endl;
        for (int i = 0; i < btpknapinc.size(); i++)
                file << btpknapinc[i].p << " ";
        file << btpknapsizeleft << endl;
        file << btpknapval << " " << btp_time_used << endl;

        file << "dynamic" << endl;
        for (int i = 0; i < dpknapinc.size(); i++)
        {
                file << dpknapinc[i].p << endl;
        }
        file << dpknapval << " " << dp_time_used << endl;

        file.close();
}

// a function which 'unsplits' a candidate solution by removing the smaller ring
// this function is for when the candidate solution is the array of rings

void uunsplit(int &k, vector<Ring> &r)
{
        for (int i = 0; i < r.size(); i++)
        {
                for (int j = i + 1; j < r.size(); j++)
                {
                        if (r[j].p == r[i].p)
                        {
                                r[i].init(r[j].p, k);
                                r.erase(r.begin() + j);
                        }
                }
        }
}

// a function which 'unsplits' a candidate solution by removing the smaller ring
// this function is for when the candidate solution is a subset of the array of rings
// ordered before the cutoff value

bool unsplit(int k, vector<Ring> &r, int &cutoff)
{
        bool merged = false;
        for (int i = 0; i < r.size(); i++)
        {
                if (i < cutoff && r[i].split)
                {
                        for (int j = i + 1; j < cutoff; j++)
                                if (i != j && r[j].p == r[i].p)
                                {
                                        r[i].init(r[j].p, k);
                                        for (int m = j; m + 1 < r.size(); m++)
                                                r[m] = r[m + 1];
                                        r.resize(r.size() - 1);
                                        cutoff--;
                                        merged = true;
                                }
                }
                else if (i >= cutoff && r[i].split)
                {
                        for (int j = i + 1; j < r.size(); j++)
                                if (i != j && r[j].p == r[i].p)
                                {
                                        r[i].init(r[j].p, k);
                                        for (int m = j; m + 1 < r.size(); m++)
```

```
                                                r[m] = r[m + 1];
                                        r.resize(r.size() - 1);
                                        merged = true;
                                }
                }
        }
        return merged;
}

// this is the main function

int main()
{
        int k;
        char filename[] = "ESF.data";
        ofstream file(filename);

        for (k = 10; k <= 300; k++)
        {
                file << k << "        ";
// begin writing the file in which the data will be stored



                double target;
                target = estimateg(k);
// get our estimate of the final answer with an extra factor of k
                makesmallprimes(k); // get the primes less than or equal to k

                vector<Ring> r;
                r.resize(primelen);

                for (int i = 0; i < primelen; i++)
                {
```

```
        r[i].init(prime[i], k); // initialize the rings
}

dpknap(r, log(target)); // before splitting rings, do dynamic program
for (int i = 0; i < primelen; i++)  // split rings
{
        Ring x;
        if (r[i].asplit_simple() < r[i].A)
        {
                x.init(r[i].p, k, r[i].asplit_simple());
                r[i].split = x.split = true;
                r[i].babya = x.a;
                r[i].makeknap();
                r.push_back(r[i]);
                r[i] = x;
        }
}

btpknap(r, log(target));    // branch and bound knapsack with pruning

btknap(r, log(target)); // branch and bound knapsack without pruning
uunsplit(k, btknapinc);

start = clock(); // start timer for greedy solution
sortbyratio(r); // sort rings

double totalsize = 0;
double totalvalue = 0;

gsplitknapinc.resize(0);
double modulus = 1;
double residues = 1;
int cutoff = 0;
```

```
        for (int i = 0; i < r.size(); i++)  // find cutoff point
        {
                if (totalsize + r[i].size <= log(target))
                {
                        totalsize += r[i].size;
                        totalvalue += r[i].value;
                }
                else
                {
                        cutoff = i;
                        break;
                }
        }
        while (unsplit(k, r, cutoff));
        ending = clock();   // end timer
        for (int i = 0; i < cutoff; i++)
        {
                gsplitknapinc.push_back(r[i]);
        }

        gsplitknapval = val(gsplitknapinc);
        gsplitknapsizeleft = log(target) - siz(gsplitknapinc);
        g_time_used = ((double)(ending - start)) / CLOCKS_PER_SEC;

        print(k, r, cutoff);    // print file with information for this k
        cout << k << endl;
        file << gsplitknapval << "        " << g_time_used << "        " <<
                btknapval << "        " << bt_time_used << "        " <<
                btpknapval << "        " << btp_time_used << "        " <<
                dpknapval << "        " << dp_time_used << endl;
                // collect data in file
}
```

```
        return 0;
}
```

6 Bibliography

References

[1] Jonathan P. Sorenson. "The Pseudosquares Prime Sieve". In: *Proceedings of the 7th International Symposium on Algorithmic Number Theory (ANTS-VII)*. Ed. by Florian Hess, Sebastian Pauli, and Michael Pohst. LNCS 4076, ISBN 3-540-36075-1. Berlin, Germany: Springer, July 2006, pp. 193–207.

[2] Jonathan P. Sorenson. "Sieving for pseudosquares and pseudocubes in parallel using doubly-focused enumeration and wheel datastructures". In: *Proceedings of the 9th International Symposium on Algorithmic Number Theory (ANTS-IX)*. Ed. by Guillaume Hanrot, Francois Morain, and Emmanuel Thomé. LNCS 6197, ISBN 978-3-642-14517-9. Nancy, France: Springer, July 2010, pp. 331–339.

[3] P. Erdős, C. B. Lacampagne, and J. L. Selfridge. "Estimates of the least prime factor of a binomial coefficient". In: *Math. Comp.* 61.203 (1993), pp. 215–224. ISSN: 0025-5718. DOI: `10.2307/2152948`. URL: `https://doi.org/10.2307/2152948`.

[4] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[5] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer Berlin Heidelberg, 2013. ISBN: 9783540247777. URL: `https://books.google.com/books?id=wmL2BwAAQBAJ`.

[6] Oscar Levin. *Discrete Mathematics: An Open Introduction*. 2019. URL: `http://discrete.openmathbooks.org/dmoi3.html`.

[7] Renate Scheidler and Hugh C. Williams. "A method of tabulating the number-theoretic function $g(k)$". In: *Math. Comp.* 59.199 (1992), pp. 251–257. ISSN: 0025-5718. URL: `https://doi.org/10.2307/2152995`.

[8] Brianna Sorenson, Jonathan P Sorenson, and Jonathan Webster. "An Algorithm and Estimates for the Erdős-Selfridge Function (work in progress)". In: *arXiv preprint arXiv:1907.08559* (2019).