I give permission for public access to my Honors paper and for any copying or digitization to be done at the discretion of the College Archivist and/or the College Librarian.

Signed_____

Rui Liu

Date _____

A Set Partition Analog of the Erdös-Szekeres Theorem

Rui Liu

Department of Mathematics and Computer Science
Rhodes College
Memphis, Tennessee

2013

Submitted in partial fulfillment of the requirements for the
Bachelor of Science degree with Honors in Mathematics

Signature Page

Contents

List of Figures

Abstract

A Set Partition Analog of the Erdös-Szekeres Theorem

by

Rui Liu

The monotonic subsequence problem has been studied in depth. Mathematicians and theoretical computer scientists have found and proved various results such as the Erdös-Szekeres Theorem and have studied algorithms to find longest monotonic subsequences. The objective of my project is to establish the counterparts of the results mentioned above in the context of set partitions, namely the heaviest free subpartition problem. More specifically, I introduce concepts such as "subpartition", "freeness" and other related terms to formulate the free subpartition problem. Then, I discuss various representations of the problem and evaluate the relationship between this problem and other widely studied algorithmic problems. Finally, I analyze the difficulties we encountered and present a "greedy" algorithm that approximates the optimal solution.

1 Background

1.1 The Longest Monotonic Subsequence Problem

In mathematics and theoretical computer science, the longest increasing (decreasing) subsequence problem is to find a subsequence of a given sequence in which the subsequence elements are in sorted order, lowest to highest (highest to lower), and in which the subsequence is as long as possible. For example, in the following sequence

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

a longest increasing subsequence is

$$0, 2, 6, 9, 13, 15$$

This subsequence has length 6, and there is not any subsequence of length 7.

It is noteworthy that this longest increasing (decreasing) subsequence is not necessarily contiguous or unique. For instance,

$$0, 4, 6, 9, 11, 15$$

is another increasing subsequence of length 6. Also, neither of these two subsequences above are consecutive subsequences.

1.2 The Erdös-Szekeres Theorem

A classic theorem of Paul Erdös and George Szekeres [3] regarding the longest increasing(decreasing) subsequence problem states that

Theorem From a sequence of $n^2 + 1$ distinct real numbers we can always extract a *monotonic* subsequence of length at least $n + 1$, and $n^2 + 1$ is the minimum length to

guarantee the existence of an monotonic subsequence of length $n + 1$.

In other words, some sequences of n^2 distinct real number do not have a monotonic subsequence of length $n + 1$. For example, the sequence

$$7, 8, 9, 4, 5, 6, 1, 2, 3$$

has many monotonic subsequence of length 3 but does not have any monotonic subsequence of length 4. The reason behind this is that in order for a sequence to have a monotonic subsequence of length $3 + 1 = 4$, the sequence should contains at least $3^2+1 = 10$ elements. However, the sequence we listed above contains only 9 elements.

Note that this theorem does not specify whether the subsequence is monotonically increasing or monotonically decreasing. The reason behind this is that both increasing subsequence or decreasing subsequence represent local orderliness among the global chaos. Moreover, if we reverse a sequence, then all the increasing subsequences become decreasing, and all the decreasing subsequences become increasing. It is not necessary to differentiate between the two. In fact, we will observe that this uniform notion of "monotone" provides a smooth transition into the study of set partitions.

2 Set Partitions

2.1 Definitions

Mathematicians and theoretical computer scientists have been studying sequences of distinct numbers for decades. Instead of considering sequences, we are more interested in the partitions of a set of positive integers. We will establish a set partition related problem similar to the classic subsequence problem described above. First of all, we will define the corresponding terms of "sequence," "subsequence," "length," and "monotone" in the context of set partition.

- A *partition* of a set is comprised of one or more *blocks*, and each element of the set falls into one of these blocks. In other word, blocks are pairwise disjoint nonempty subsets of the set, and the union of these subsets equals the set itself. We use "/" to separate blocks from each other. For example, $\pi = 138/24/579/6$ is a partition of the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. By convention we list the elements of each block in increasing order and arrange the blocks in increasing order of their least elements.

- If we take nonempty subsets of one or more blocks, then we call the collection of these subsets of blocks a *subpartition* of the original set partition. For instance, both $\pi_1 = 18/4/57$ and $\pi_2 = 24/6/9$ are subpartitions of $\pi = 138/24/579/6$. Notice that subpartition is the counterpart of "subsequence" in set partition. Furthermore, the subsets of blocks of the original partition are now the blocks of this subpartition.

- For both a partition and a subpartition, we call the sum of its blocks' cardinalities its *weight*. If the weight of partition π_A is larger than that of π_B, we say that π_A is "heavier" than π_B. For example, $\pi = 138/24/579/6$ has weight 9. $\pi_1 = 18/4/57$ has weight 5 and $\pi_2 = 24/6/9$ has weight 4. π_1 is heavier than π_2.

Weight is the corresponding concept of "length" in set partition; both weight and length represent the "size" of the mathematical objects of interest.

- A *link* of a partition is a 3−subpartition of the form a_1a_2/b, where $a_1 < b < a_2$, a_1, a_2 belong to the same block and b is in a different block. For the sake of convenience we usually express link ac/b as abc. A partition or subpartition is "13/2-*free*," "*link-free*" or simply "*free*," if it does not contain any links. In other words, a partition/subpartition is free if and only if for any two numbers a, b in a block, there is no element c in another block satisfying $a < c < b$. For instance, $\pi_2 = 24/6/9$ is free, but $\pi_3 = 18/4/57$ is not free since it contains links 148 and 158 and 178.

In addition, given a set partition π we call the collection of all links in π the linkset $\mathcal{L}$ of π. Professor Gottlieb has come up with the following axioms that all linksets $\mathcal{L}$ need to satisfy:

1. If $abc \in \mathcal{L}$ then
   (a) for all $d \in [n]$ we have $adb \notin \mathcal{L}$.
   (b) for all $d \in [n]$ we have $bdc \notin \mathcal{L}$.
   (c) for all $d \in [n]$ we have $dac \notin \mathcal{L}$.
   (d) for all $d \in [n]$ we have $acd \notin \mathcal{L}$.
2. If $abc \in \mathcal{L}$ and $cde \in \mathcal{L}$ then $abe \in \mathcal{L}$ and $ade \in \mathcal{L}$
3. If $abc \in \mathcal{L}$ and for all $e \in [n]$ we have $aed \notin L$ and $dec \notin l$ then $adc \in L$.

Now we have all the corresponding terms of the longest monotonic subsequence problem to establish its counterpart in the context of set partitions. We will first study the following computational problem.

The heaviest free subpartition problem is to find a subpartition of a given set partition in which the subpartition contains no link and in which the subpartition is as heavy as possible.

2.2 The Number of Set Partitions

It is beneficial to look at how the number of partitions grows as the weight increases. We can easily compute and enumerate that there are 5 partitions of the set $\{1, 2, 3\}$, namely 1/2/3 and 12/3 and 1/23 and 13/2 and 123. It is also not so hard to count that there are 15 partitions of $\{1, 2, 3, 4\}$. How about for even larger sets? Mathematicians have been working on the number of partitions for decades. The total number of set partitions of a given weight are commonly known as *Bell number* [4]. We shall denote Bell numbers by ϖ_n. The first few cases are

n = 0	1	2	3	4	5	6	7	8	9	10	11	12
ϖ_n = 1	1	2	5	15	52	203	877	4140	21147	115975	678570	4213597

The Bell numbers ϖ_n for $n \geq 0$ must satisfy the recurrence relation

$$\varpi_{n+1} = \varpi_n + \binom{n}{1}\varpi_{n-1} + \binom{n}{2}\varpi_{n-2} + \ldots = \sum_k \binom{n}{k}\varpi_{n-k}$$

The reason behind this recursive formula is that every partition of $\{1, \ldots, n+1\}$ is formed by choosing k elements of $\{1, \ldots, n\}$ to put in the block containing $n+1$ and by partitioning the remaining elements in ϖ_{n-k} ways, for some k. Notice that this sequence grows very rapidly, but not as fast as $n!$. Knuth has proved that $\varpi_n = \Theta((n/\log n)^n)$ [8].

3 Representations of A Set Partition

We introduce various ways to represent a set partition. Each of these representations demonstrates some aspects of the combinatorial objects.

3.1 Restricted Growth Functions

One of the most common and convenient ways to represent a set partition inside a computer is to encode it as a *restricted growth function* or *restricted growth string*, or RGF for short [9]. A restricted growth function is a string $a_1a_2\ldots a_n$ of nonnegative integers in which we have

$$a_1 = 1 \text{ and } a_{j+1} \leq 1 + \max(a_1, \ldots, a_j) \text{ for } 1 \leq j < n.$$

The main idea of this encoding is to set $a_j = a_k$ if and only if j and k are in the same block, and to choose the smallest available number for a_j whenever j is smallest in its block. In other words, what restricted growth function does is to record the block index of each number in the partition. For instance, the restricted growth function for $\pi = 138/24/579/6$ is 121234313. Since the number 1 appears in the first block, we put 1 in the first position of the restricted growth function. The number 2 appears in the second block, so the second position is 2, and the rest of the digits follow the same pattern.

A sequence s of $\mathbb{N}$ is called *clumpy* if it satisfies that if $i < t < j$ and $s(i) = s(j)$, then $s(i) = s(t) = s(j)$. We now show that

Theorem A partition π is free if and only if its RGF s_π is clumpy.

Proof: If π is free, then for all i, j such that i and j are in the same block, we know that $s_\pi(i) = s_\pi(j)$. Suppose s_π is not clumpy, then we know that there exists t such that $i < t < j$, but $s_\pi(i) \neq s_\pi(t)$. This also means that t is not in the same block as i and j and $i < t < j$. Obviously itj is a link and π is not free. We have a contradiction, then our assumption is false. Therefore s_π is clumpy.

On the other hand, if s_π is clumpy, then we know that for all i, t, j, if $i < t < j$ and $s_\pi(i) = s_\pi(j)$, then $s_\pi(i) = s_\pi(t) = s_\pi(j)$. Suppose π is not free, then we know that there exists p, q, r such that $p < q < r$ and p, r are in the same block, but q is not. Then by the formation of a restricted growth function, we know that $s_\pi(p) = s_\pi(r) \neq s_\pi(q)$, which is contradictory to our claim that s_π is clumpy. Therefore our assumption is false and π is free. □

George Hutchinson [6] has developed the following algorithm to generate the restricted growth functions in lexicographic order.

Algorithm H (*Restricted growth strings in lexicographic order*). Given $n \geq 2$, this algorithm generates all partitions of $\{1, 2, \ldots, n\}$ by visiting all strings $a_1 a_2 \ldots a_n$ that satisfy the restricted growth condition in the definition listed above. We maintain an auxiliary array $b_1 b_2 \ldots b_n$, where $b_{j+1} = 1 + \max(a_1, \ldots, a_j)$; the value of b_n is actually kept in a separate variable, m, for efficiency.

H1. [Initialize.] Set $a_1 \ldots a_n \leftarrow 0 \ldots 0$, $b_1 \ldots b_n \leftarrow 1 \ldots 1$, and $m \leftarrow 1$.

H2. [Visit.] Visit the restricted growth function $a_1 \ldots a_n$, which represents a partition into $m + [a_n = m]$ blocks. Then go to **H4** if $a_n = m$.

H3. [Increase a_n.] Set $a_n \leftarrow a_n + 1$ and return to **H2**.

H4. [Find j.] Set $j \leftarrow n - 1$; then, while $a_j = b_j$, set $j \leftarrow j - 1$.

H5. [Increase a_j] Terminate if $j = 1$. Otherwise set $a_j \leftarrow a_j + 1$.

H6. [Zero out $a_{j+1} \ldots a_n$] Set $m \leftarrow b_j + [a_j = b_j]$ and $j \leftarrow j + 1$. Then, while $j < n$, set $a_j \leftarrow 0, b_j \leftarrow m$, and $j \leftarrow j + 1$. Finally set $a_n \leftarrow 0$ and go back to **H2**.

In Algorithm H, step H1, H2, ..., H6 are performed $1, \varpi_n, \varpi_n - \varpi_{n-1}, \varpi_{n-1}, \varpi_{n-1}$ $\varpi_{n-1} - 1$ times respectively.

An obvious drawback for the restricted growth function to represent a set partition is that we will lose some information when taking the restricted growth function of a subpartition. To be more specific, consider the partition 1/2345/6, whose RGF

representation is 122223. All of subpartition 1/2/6 and 1/3/6 and 1/4/6 and 1/5/6 have RGF representation 123. This will cause potential ambiguity. To fix this problem, when taking subpartition of a partition, instead of trying to compute the RGF for the new subpartition, the only changes we do to the RGF are to change the value of position i to be 0, if i is a number that is not in the subpartition. For instance, now 1/2/6 will have 120003 as its representation, since we have deleted number $3, 4, 5$ and we replace their positions with 0. Note that the representation of a subpartition may not be a restricted growth function any more and this representation only has meaning when we are considering a subpartition of a given partition. For example, 14/23 has RGF as 1221, however its subpartion 23/4 has representation 221, which is not a RGF.

The heaviest free subpartition problem becomes the problem to find the longest clumpy subsequence of a restricted growth function that represents a set partition.

3.2 The Interval Scheduling Problem

A set partition can also be represented as a variation of the scheduling problem. The basic idea to use intervals on the same level to denote numbers in the same block, and numbers in two different block will appear as intervals on different levels in the graph.

A formal definition of the interval scheduling problem is: We have a set of requests $\{1, 2, \ldots, n\}$; the i^{th} request corresponds to an interval of time starting at $s(i)$ and finishing at $f(i)$. We say that a subset of the requests is compatible if no two of its members overlap in time, and our goal is to accept as large a compatible subsets as possible. A compatible set of maximum size will be called *optimal* [7].

Since we usually place these intervals into different height when illustrating them in the 2-D plane. For each 2-D plane that represent a set partition, we place the intervals representing two numbers in the same level if and only if these two numbers

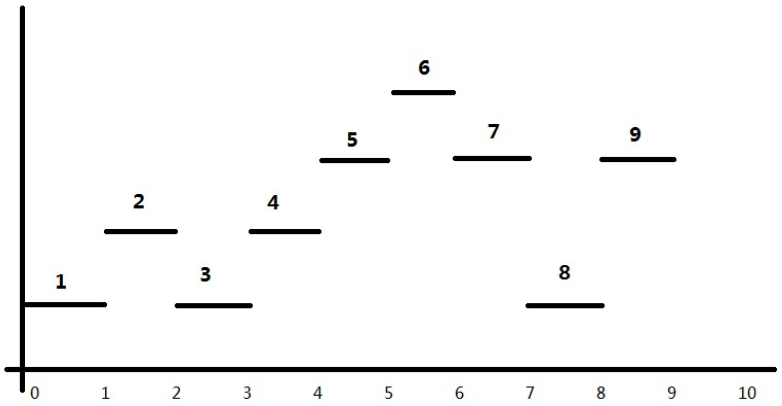

Figure 1: The interval representation of the partition 138/24/579/6.

are in the same block. Figure 1 is an illustration of $\pi = 138/24/579/6$.

Now, the goal of the problem is to find a special optimal compatible set of intervals. It is a "path" from left to right such that once we leave a level, we will never re-enter the same level.

3.3 The Weighted Scheduling Problem

Another variation of the interval scheduling problem can also provide some insights in to the structure of the heaviest free subpartition problem. The weighted scheduling model is similar to the generic interval scheduling problem discussed above. The main difference is that the each intervals in the 2-D plane now has a weight. Most importantly, the length of each interval demonstrated in the 2-D plane is not related to its weight. Figure 2 is an example in which the weight v_k of an interval k is completely irrelevant from its length.

A possible representation of the weighted scheduling problem is to use each interval to represent a combination of numbers from a block and the weight of this interval is the sum of the number selected. And when we have all these intervals, we need to find a way to place them in the 2-D plane such that if two combinations of numbers

cannot coexist (because they will cause a link to appear) then the two intervals that represents them will overlap. After this process, the heaviest free subpartition problem is reduced into the weighted scheduling problem in which the goal is to find a sequence of non-overlapping intervals such that the sum of the weight is maximized.

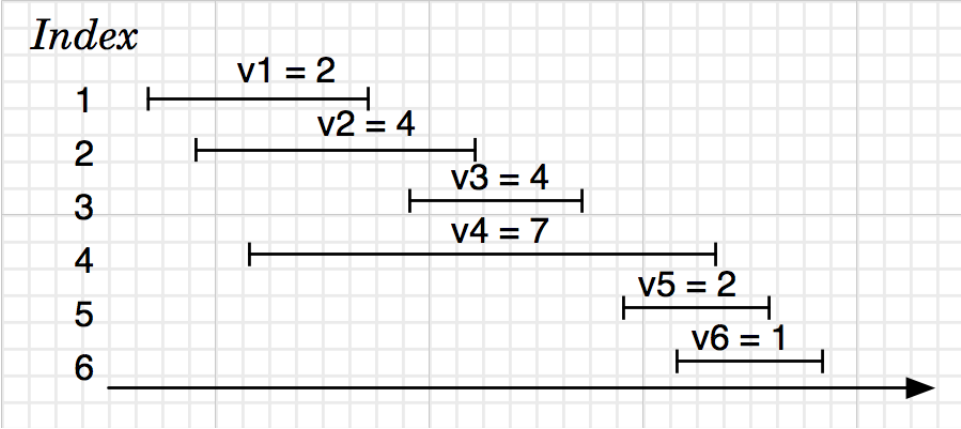

Figure 2: An example of the weighted scheduling problem in which the weight of an interval is not related to its length.

This representation is tempting because it carries the concept of the weight naturally. Additionally, it is elegant even to think of the idea that we are using the relative positions in the 2-D plane to denote the links. A $O(n \log n)$ algorithm using dynamic programming has been invented to solve the weighted scheduling problem [7]. It is a pity that we have not found a way to arrange the intervals described above. Such a way of arrangement is expected to run in exponential-time even if it is found.

3.4 The Hypergraph Vertex Cover Problem

A $k-$uniform hypergraph $H = (V, E)$ consists of set of vertices V and a collection E of $k-$element subsets of V called hyperedges. A *vertex cover* of H is a subset $S \subseteq V$ such that every hyperedge in E intersects S, in other word, $e \cap S \neq \emptyset$ for each $e \in E$. The Ek-Vertex-Cover problem is the problem of finding a minimum size vertex cover in a k-uniform hypergraph [2]. This problem is alternatively called the

minimum hitting set problem with sets of size k. Figure 1 illustrates an example of the Ek-Vertex-Cover problem and a vertex cover of minimum size.

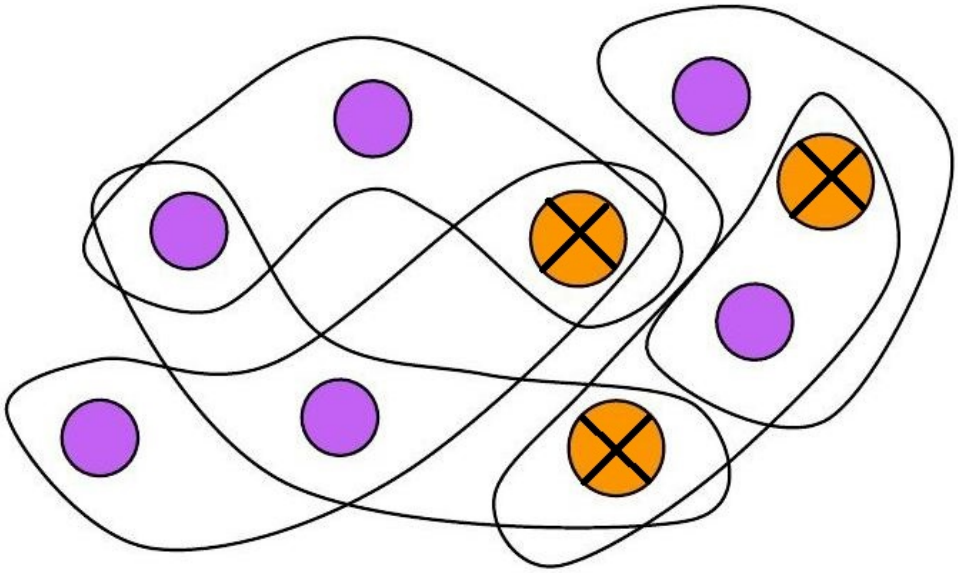

Figure 3: This is a hypergraph and the set of vertices marked with "X" is a minimum vertex cover.

To convert the heaviest free subpartition problem using this model, let V be the set of numbers in the partition π and let E be the linkset of π, i.e. E contains all the links of π. Since we define link as a triple of numbers in the partition, then we know that in this case $k = 3$, and the new hypergraph $H = (V, E)$ we just defined is a 3-uniform hypergraph.

Now we finally have an elegant and simple way to transform our problem into a widely studied one. However, the Ek-Vertex-Cover problem is a fundamental NP-hard optimization problem. Being an NP-hard problem means that this problem is "at least as hard as the hardest problems in NP". Although the properties we tracked through studying the linkset will help us simplify the hypergraphs we are dealing with, the problem is still hard. Since the Ek-Vertex-Cover problem on a generic hypergraph is NP-hard, theoretical computer scientists' efforts on this problem are mainly applying approximation algorithms. Dealing with approximation algorithms

is beyond the scope of this project.

4 Algorithms

4.1 The Dynamic Programming Approach

The application of dynamic programming requires the existence of a clear relation between the problem's solution and its subproblems' solution. In other words, the dynamic programming approach might apply if the problem exhibits *optimal substructure* in which an optimal solution to the problem contains within it optimal solutions to subproblems [1]. A classic example is the algorithm to find the longest increasing subsequence. By applying dynamic programming, the longest increasing subsequence problem can be solved in $O(n \log n)$ time [5]. However, we are unable to find an optimal substructure in the heaviest free subpartition problem. Let π be a set partition of weight n and π_1 be its subpartition of weight $n-1$. From our observation, there is no certain relation between the heaviest free subpartition of π and that of π_1.

Since we faced seemingly unresolvable obstacles in the direction of dynamic programming, we turned out emphasis on another widely used technique, the greedy algorithm.

4.2 The Greedy Approach

We have come up with an algorithm to find a free subpartition in a given partition using the greedy algorithm design paradigm. By calling an algorithm "greedy" we mean that at each step of the decision process, we always pick the locally optimal choice with the hope of finding a global optimum. The designers of the greedy algorithms always need a heuristic to make the choice. This heuristic is an estimation of the structure we are exploring and evaluations of the possible choices.

4.2.1 The Naive Greedy Approach

For the heaviest free subpartition problem, the heuristic we choose is the appearances of each number in the linkset $\mathcal{L}$. More specifically, we wil do the following:

1. Create an array $f[\ ]$ of size n and initialize all $f[1], \ldots, f[n]$ to be 0.

2. Then we visit all the triples in the linkset $\mathcal{L}$, every time we encounter number i, we increase $f[i]$ by 1. When we have finished this process for all the links, the value of $f[i]$ is the number of i's appearances in the linkset $\mathcal{L}$.

3. One natural way to modify a partition so that it is free is to break the links; when there is no link left, the partition is free. At each step, it is natural to delete the number that appears most frequently in the linkset so that the amount of links we break will be maximized at this step. So at each step we choose the number to delete to be the i such that $f[i] = \max\{f[1], \ldots, f[n]\}$. If there are more than one such i, we always choose the least one, which is why we call this the "naive" approach.

4. Then we remove all the links that contains i and update the array $f[\ ]$.

5. We repeat step 3-4 until there is no links left.

The partition we are left with now is a free subpartition and we hope by the greedy assumption that this is a heaviest free subpartition. We call the algorithm described above as the *naive greedy algorithm*.

For example, Consider the partition $15/238/46/7\overline{10}/9$ whose RGF is 1223134254.

- Now the content of f array is

n =	1	2	3	4	5	6	7	8	9	10
$f[n]$ =	3	5	5	4	6	3	4	9	1	2

- We delete 8.
- Now the content of f array is

n =	1	2	3	4	5	6	7	8	9	10
$f[n]$ =	3	1	1	2	4	1	1	0	1	1

- We delete 5.
- Now the content of f array is

n =	1	2	3	4	5	6	7	8	9	10
$f[n]$ =	0	0	0	0	0	0	1	0	1	1

- We delete 7
- Now the content of f array is

n =	1	2	3	4	5	6	7	8	9	10
$f[n]$ =	0	0	0	0	0	0	0	0	0	0

- Done! The resulting subpartition is $1/23/46/9/\overline{10}$ and this is a heaviest free subpartition.

In fact it is not so hard to find partitions such that this naive greedy approach fails.

For instance, consider the partition $159/23/468\overline{10}/7$ whose RGF is 1223134313.

- Now the content of f array is

n =	1	2	3	4	5	6	7	8	9	10
$f[n]$ =	9	2	2	8	9	6	6	6	12	6

- We delete 9.
- Now the content of f array is

n =	1	2	3	4	5	6	7	8	9	10
$f[n]$ =	3	1	1	6	6	3	4	3	0	3

- We delete 4.
- Now the content of f array is

n = 1	2	3	4	5	6	7	8	9	10
$f[n] = 2$	1	1	0	2	2	2	1	0	1

- We delete 1
- Now the content of f array is

n = 1	2	3	4	5	6	7	8	9	10
$f[n] = 0$	0	0	0	0	2	2	1	0	1

- We delete 6
- Done! The free subpartition we get from the greedy algorithm is $5/23/8\overline{10}/7$.
- However, there is an even heavier free subpartition $1/23/468\overline{10}$

Suppose the greedy algorithm returns us a free subpartition of weight $n - t$, in other word, the greedy process deletes t elements from the original partition. They method we used to check if the result of the greedy algorithm is the heaviest free subpartition, is by trying to delete all the $(t-1)-$combinations of numbers in $[n]$ from the original partition and test if any of the resulting subpartition is free. If there is one such subpartition that is free, then result of the greedy algorithm is not optimal, and the greedy algorithm fails. The following algorithm [8] to generate all the combination is used when testing the quality of result of the greedy approach.

Algorithm L (*Lexicographic combinations*).
This algorithm generates all $t-$combinations $c_t \dots c_2 c_1$ of the n numbers $\{0, 1, \dots, n-1\}$, given $n \geq t \geq 0$. Additional variables c_{t+1} and c_{t+2} are used as sentinels.

L1. [Initialize.] Set $c_j \leftarrow j - 1$ for $1 \leq j \leq t$; also set $c_{t+1} \leftarrow j - 1$ and $c_{t+2} \leftarrow 0$.

L2. [Visit.] Visit the combination $c_t \dots c_2 c_1$

L3. [Find j.] Set $j \leftarrow 1$. Then, while $c_j + 1 = c_{j+1}$, set $c_j \leftarrow j - 1$ and $j \leftarrow j + 1$; eventually the condition $c_j + 1 \neq c_{j+1}$ will occur.

L4. [Done?] Terminate the algorithm if $j > t$.

L5. [Increase c_j.] Set $c_j \leftarrow c_j + 1$ and return to **L2**.

Since we mentioned that an the Ek-Vertex-Cover Problem, which can be viewed as a generalization of the heaviest free subpartition problem, is NP-hard, a natural question to ask is "Is the free subpartition problem in NP?" A problem is in NP if and only if given a solution, we can verify whether this solution is valid or not in polynomial-time. Although being NP does not tell us sufficient information about the algorithm to find such a solution, failure to be NP will definitely demonstrate the complexity of the problem. To determine whether there is a polynomial-time verifier for the heaviest free subpartition problem, it is necessary to have a close look at the growth rate of $\binom{n}{k}$.

For $k = 1$,

$$\binom{n}{1} = n \in \Theta(n)$$

For $k = 2$,

$$\binom{n}{2} = \frac{n(n+1)}{2} \in \Theta(n^2)$$

For $k = 3$,

$$\binom{n}{3} = \frac{n(n-1)(n-2)}{6} \in \Theta(n^3)$$

Because of the symmetry of binomial coefficient, the largest k we need to consider is

For $k = \frac{n}{2}$

$$\binom{n}{\frac{n}{2}} = \frac{n!}{((n/2)!)^2}$$

By applying the Stirling Formula

$$n! \sim \sqrt{2\pi n}(\frac{(n)}{e})^n$$

we then get

$$\binom{n}{n/2} \approx \frac{\sqrt{2\pi n}(\frac{(n)}{e})^n}{[\sqrt{2\pi n/2}(\frac{(n)}{2e})^{n/2}]^2} = \frac{2}{\sqrt{2\pi n}}2^n \in \Theta(2^n n^{-0.5})$$

which tells us $\binom{n}{k}$ grows exponentially near $k = \frac{n}{2}$. As a conclusion, we cannot even test the correctness of the greedy algorithm's result efficiently when the amount of numbers we delete during the greedy algorithm increases. This implementation basically becomes useless when we are testing partition of a larger weight, in which both n growth and the number we delete from the partition is possibly close to $\frac{n}{2}$. If applying Algorithm L and our current method to examine L's result, the heaviest free subpartition problem becomes NP-Hard as the weight of the partition grows.

4.2.2 A "Complete" Greedy Approach

Now since this problem is hard based on our current techniques and the computation is expensive even for the naive approach, I am less concerned about efficiency. Instead, my primary computational question becomes

for any partition, is there a sequence of greedy choices that will lead to a heaviest free subpartition?

To answer this question, I modified the greedy algorithm described above. When facing a tie in the value of $f[i]$, instead of choosing the least number i, I try all such

i's. If we follow this pattern, we are actually exploring a tree, in which each child of a node represents a locally optimal choice. We choose Depth First Search to provide us a way to traverse this "greedy" tree. We call this complete search of the greedy path tree the *complete greedy algorithm*.

Below is the pseudo-code of a depth first search using stack.

```
DEPTH-FIRST-SEARCH(G, ROOT)
1  S ← A Stack
2  S.push(root)
3  while S is not empty
4        do v ← S.pop()
5           mark node v as visited
6           for each node w that is directed from v:
7           if w is not visited:
8              then S.push(w)
```

The result of performing the "complete" greedy algorithm shows that

1. For some partitions although the naive greedy algorithm fails, there is a path in the greedy path tree that will lead to a heaviest free subpartition.

2. For some partition, no greedy path will lead to a heaviest free subpartition.

For example, the free subpartitions that the complete greedy algorithm generates for partition $12468/3/59/7\overline{10}$ are: $124/59/\overline{10}$ and $12/3/59/\overline{10}$ and $12/3/5/7\overline{10}$

While the only heaviest free subpartitions of this partition is: $12468/9/\overline{10}$

Note that all the free subpartitions that formed by the greedy algorithm delete the number 6 and 8, while the real heaviest free subpartition does not delete neither of them. Here the locally optimal step do not accumulate to a global optimum. Thus, we can conclude that the failure of the greedy algorithm shows us that the array $f[\,]$ is not sufficient to represent the actual quality of each choice.

4.2.3 Evaluation of Performance

Nevertheless, both the greedy and the complete greedy algorithm can provide a good approximate of the heaviest free subpartition. By implementing both algorithms, running them on all the set partition of weight 6, 7, 8, 9, 10, 11 and computing the percentage of the partitions on which the greedy algorithm works in all the partitions that are not free already, we found that the improvement of the complete greedy algorithm compared to the naive approach is not that significant, as we can see in the table below.

For the naive greedy algorithm

weight	6	7	8	9	10	11
% of success	100%	99.51%	98.70%	97.62%	96.42%	95.08%

For the complete greedy algorithm

weight	6	7	8	9	10	11
% of success	100%	100%	99.80%	99.30%	98.64%	97.89%

5 A Ramsey Theoretic Theorem

The ultimate goal of this project is to answer the following question:

> How large must the minimum value of n be in order to guarantee that every partition of the set $\{1 \dots n\}$ has a free subpartition of weight k?

We call this minimum value of n $f(k)$, since it is clearly a function of k. First of all, does such $f(k)$ even exist? The answer is positive. Consider a set of cardinality $(k-1)^2+1$. When using the convention of set partition described above, each partition of the set could be expressed as a sequence of length $(k-1)^2+1$. For example, 157/246/389 has sequence representation $(1,5,7,2,4,6,3,8,9)$. By the Erdös-Szekeres Theorem, there will be a monotonic subsequence of length k. These k numbers and the "/" between them in the original sequence form a sequence representation of a free subpartition of weight k. This application of the Erdös-Szekeres Theorem shows that all the partitions of $|(k-1)^2+1|$-set have a free subpartition of weight k. This implies the existence of $f(k)$ and $f(k) \leq (k-1)^2+1$.

Professor Gottlieb conjectures that $f(k) = \lfloor \frac{(k+1)^2}{4} \rfloor$. If we complete the proof of this conjecture, it will be a significant leap from the loose upper bound $(k-1)^2+1$. Like the study of the longest monotonic subsequence, this fundamental problem of my project is a concrete example of Ramsey Theory.

Professor Gottlieb has also shown that there exist partitions of weight $\left\lfloor \frac{(n+1)^2}{4} \right\rfloor - 1$ that do not have a free subpartition of weight n. The remaining portion to prove this conjecture is to show that

A partition of weight $\left\lfloor \frac{(n+1)^2}{4} \right\rfloor$ must have a free subpartition of weight n.

I have proposed a proof of this direction which turned out to be false. When facing the obstacles in directly proving the result, I also tried to verify the existence of free n-subpartition in a $f(n)$-partition. Both Professor Gottlieb and I have been

able to verify the correctness when $n \leq 6$, that is all partitions of weight 12 has a free subpartition of weight 6. The method I used to check the existence of such free subpartition is by applying the naive greedy algorithm as an approximation first and then check the heavier free subpartition to see if its weight is greater than or equal to n. So far, to test the case $n = 6$, my program takes around 15 minutes. However, since $\varpi_{16}/\varpi_{12} \geq 2400$, I haven't been able to verify the case $n = 7$ yet.

6 Conclusion

This problem is, in fact, much harder than we previously expected. For finding the heaviest free subpartition, there is no obvious way to apply the dynamic programming approach, while the greedy algorithms can only serve as an approximation for small weight. We also faced difficulties in the verification of the greedy algorithm due to our current lack of computing power.

Although this project yields no essential results at this point, I have spent a considerable amount of time and energy on coding, reading articles and considering many possible directions. The knowledge I gained and the skills I have sharpened cannot be shown completely through this report. I will keep working on this problem when I graduate, because it shows me more complexity and attractions as I explore.

Special thanks to Professor Eric Gottlieb for his yearlong support and guidance.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[2] Dinur, I., Guruswami, V., Khot, S., and Regev, O. A new multilayered pcp and the hardness of hypergraph vertex cover. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing* (New York, NY, USA, 2003), STOC '03, ACM, pp. 595–601.

[3] Erdös, P., and Szekeres, G. A combinatorial problem in geometry. In *Classic Papers in Combinatorics*, I. Gessel and G.-C. Rota, Eds., Modern Birkhäuser Classics. Birkhuser Boston, 1987, pp. 49–56.

[4] Erickson, M. J. *Introduction to Combinatorics*. Wiley-Interscience, 1996.

[5] Hunt, J. W., and Szymanski, T. G. A fast algorithm for computing longest common subsequences. *Commun. ACM 20*, 5 (May 1977), 350–353.

[6] Hutchinson, G. Partioning algorithms for finite sets. *Commun. ACM 6*, 10 (Oct. 1963), 613–614.

[7] Kleinberg, J., and Tardos, E. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[8] Knuth, D. E. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional, 2005.

[9] Stanton, D., and White, D. *Constructive Combinatorics*. Undergraduate Texts in Mathematics. Springer-Verlag, 1986.