Group 4 - Final Report

Team Members:

- Batuhan Yıldırım 26478
- Berna Yıldıran 26431
- Kerem Kurnaz 26586
- Semih Zaman 28358

1. Problem Description

Formal Problem Description:

Input of the problem consists of:

Collections C and D which is a set of finite sets,

$C = \{C_1, C_2, ..., C_n\}$

$D = \{D_1, D_2, ..., D_n\}$

and let $D \subset C$

The problem is aimed to find D with the largest cardinality such that,

$\forall i, j \in N,\ 1 \leq i < j \leq k,$

D_i and D_j have no common elements i.e. $D_i \cap D_j = \emptyset$?

Intuitive Description:

There exists a finite set C and, C has a subset D.

In this project it is aimed to find a subset D with maximum cardinality where subsets in D are pairwise disjoint.

An instance of the problem:

$S = \{\{1, 2, 3\}, \{3, 4\}, \{5, 6, 7\}, \{8, 9\}, \{12\}, \{17, 18, 19\}\}$

P_i consists of both mutually disjoint and non-disjoint subsets. We do not need to consider the sets with only 1 element since these sets are trivial.

$P_1 = \{\{1,2,3\},\{3,4\},\{5,6,7\},\{8,9\}\}$

$P_2 = \{\{1,2,3\},\{5,6,7\}\}$

$P_3 = \{\{1,2,3\},\{5,6,7\},\{8,9\},\{17,18,19\}\}$

In this example since we are looking for pairwise disjoint sets we should not consider P_1. Since P_2 and P_3 consists of pairwise disjoint sets and $|P_2| < |P_3|$, thus the correct solution would be P_3.

In this example since we are looking for pairwise disjoint sets we should not consider P_1. Since P_2 and P_3 consists of pairwise disjoint sets, among these two sets it can be stated that $|P_2| < |P_3|$. Therefore it can be concluded that among the pairwise disjoint sets P_2 and P_3, P_3 is our solution since it is larger than P_2.

Possible Applications

An example for the possible applications set packing problem is planning the exam scheduling in universities. In order to exams to be completed successfully, none of the exams should overlap with each other and a student should be able to participate in all of her/his exams without any problem.

For the purpose of being efficient in exam scheduling, student resources should organize maximum number of exam within a specific time period, between 08:30AM - 10:30AM, considering the fact that the duration of an exam in this university is 2 hours. In order to find this optimal scheduling, all possible sets of exam schedules should be listed in a set and from this set, the disjoint subset which consists of maximum number of elements should be found out. This real life problem which aimed to find maximum amount of exams that can be done within a specific time period utilizes set packing to solve this problem.

Set packing used in many other applications such as airport staff - flight scheduling, and determining facility locations. Set packing is also used in fields such as communication networks and computer vision to solve diverse problems.

NP-Completeness of the Set Packing Problem

Our claim: The Set Packing Problem that we are dealing with is an NP-hard problem.

Proof: To prove that the Set Packing Problem is NP-Hard, we take the help of the Clique Decision Problem that is already NP-Hard and show that this problem can be reduced to the Set Packing problem (Karp, 1972).

Let assume that every vertex corresponds to a different set such that any two sets are disjoint if there does not exist an edge between them in the graph. It can be done like that:

Let's give different weights for each edge and then, we can insert these weights to the corresponding set which has those edges while it was a vertex. After this process, the number of sets will be $|V|$ and the total number of insertion operations will be $2|E|$. While the Clique Decision Problem is to determine if the graph contains a clique of size K, given a graph $G(V, E)$ and an integer K (Clique is a subgraph which is a complete graph in a graph). Now the problem becomes to determine whether it is possible to find any K disjoint sets between those $|V|$sets (Schreiber & Migler, 2021).

The problem we managed to get by reduction is the decision version of Set Packing Problem as follows:

Is there a collection of at least K subsets such that no two subsets intersect?

However, our problem is an optimization problem, which requires a polynomial number of calls to the decision problem. Therefore, if the K value is iterated from 1 to the $|V|$ and this decision problem is called $|V|$ times in the worst case, it corresponds to our Set Packing Problem now.

As a result, we manage to reduce an NP-Hard problem to our main Set Packing with the complexity $O(|V|(|V| + 2|E|))$ where $|V|$ represents the number of vertices and $|E|$ represents the number of edges.

Since Set Packing Problem is shown to be at least as hard as Clique Decision Problem, Set Packing Problem is also an **NP-Hard problem**.

2. Algorithm Description

A. Brute Force Algorithm

Description

This algorithm has two major parts which they finding combinations and checking those combinations. The approach to the problem is basically trying all possible cases and checking them in a logical manner to reach a solution. Logical manner states for making

code faster and eliminating an important amount of cases most of the time. However, in the worst case, we have to find and check every possible solution.

The first major part of the algorithm is finding combinations. This subproblem uses bit strings to find every possible combination of a given number of sets. In a classical manner, we have to find every case explicitly. But we know counting is also a combination of numbers and base 2 numbers are a good representation of existing and not existing. Thus we do not even have to calculate any combination but we can increase the number by 1 to get every combination. For the sake of second part elimination, this code calculates the given number of combinations of the set. For example, we need N chooses 2 which means we need every bit string combination which has only two bits are 1. We are choosing 1, 2, 3, …, n where n denotes the length of S.

Finding this combination can be seen as a dynamic programming problem since we need prior combination solutions until reaching n. This part first creates a template of zeros to generate succeeding combinations. After obtaining the template we can keep building the combinations. For example, if we have “” as a template only possible iteration can be adding “0” or “1” at the end of the string. Then we have “0” and “1” in our list. Let us keep applying this process one more step. We will have “00” and “01” from the first element, “10” and “11” from the second element. So we have every combination of bit strings for length 2. We are adding those elements in a list to remember and if we iterate n steps we will have every possible bit string combination for length k bit string. The key point here is to classify them with respect to the number of 1s in the string. This operation is done by the inner for loops and this is the first major part of the algorithm.

Now we have a bit string representation of every possible subclass of S and we need to check if those subclasses are meeting our conditions. We are looking for a set that has the highest cardinality while not having any intersecting elements in sets. Then if we start looking from k chosen k to k chosen 1, we can stop immediately if we find any subset meets the conditions since we check every set which has higher cardinality. This operation is executed in the main function and passes those combination representations to a checkout function with a translated array. This array is a

representation of a given set in array strings (not a string array even though they are also string array). The main logic here is not to spend too much time checking set intersections. So we are translating those inputs.

In vectorTranslator function, we are finding max and min elements of S and the element of S which has the biggest cardinality. Then we are creating a string of zeros in max element length and make 1 this array's elements if the position corresponding is in the array. This will allow us to check columns and decide if there is an intersection in the set. If the column sum is greater than 0 then there is an intersection. This operation can be more efficient by using int arrays and matrices instead of using strings.

In checkset function, we are choosing the vectors from translated if the corresponding bit of comb is 1. After that, we need to check if the given set of vectors are meeting the condition or not with isvalid function. isvalid function basically iterates over the columns and calculates the summation of the column. If code finds out any column is greater than 1 it terminates with a false. Without checking the rest of the array. If the code can not find any column sum greater than 1 then it returns a true. This true is in the termination step of the code and where we print our result. We are printing the number I which is the k chooses I in the loop.

Pseudocode

```
def bit_combinations:

    create bitstr templates

    append 1 to every element in this template

    append 0 to every element in this template

    make conversation to have a clearer comb table

    return lean comb table

comb_table = bit_combinations:

translated_set = translate(input)

for every combination from end to start:
```

```
    if comb is valid with translated_set:

        return true
```

B. Greedy Search Algorithm

Description

The problem is solved in brute force in the previous report with high complexities and for this report, there is a greedy algorithm to find still good results in shorter duration, for bigger vector sets, and with less space capacity. The nature of the greedy algorithm allows us to find fairly good approximations with reasonable resources.

Given the problem is asking for the largest cardinality subset where there is no intersection between sets. A basic logic can tell that if we choose set S from given sets, where size of set S is decreasing the possibility of having higher N is increases, N is the number of not intersecting sets with chosen set S. The greedy approach should make the best move with respect to its heuristic function which is looking for a local maximum. So in this case our heuristic is choosing the smallest sized set to increase our chance to obtain the highest cardinality subset.

In order to perform this operation efficiently, we are sorting the given setlist with respect to their size in ascending order. The next operation is choosing the first element of this setlist. The following step is adding the next set and checking if the validation of chosen setlist. If validation is not satisfied then we are popping the added element and keep doing the same operation for the rest of the list. In the end, we have our greedy solution for the problem where we are traversing the list only once to find a solution if we do not consider sorting.

After all this process the greedy algorithm is ready.

Pseudocode

```
translated_set = translate(input)

sort input set

define resultlist

resultlist.append(inputset[0])

For input set:
    resultlist.append(inputset_instance)
    if resultlist is not valid:
        resultlist.pop
return resultlist
```

Ratio Bound

Let us assume that $C' = \{C'_1, C'_2, \dots C_n'\}$ is the sorted version of our given input, $|C'_1| \rightarrow m_1$, and the optimal solution R (However, there can be many optimal solutions set with the cardinality of R). According to our algorithm, C'_1 is inserted into the greedy solution set at the beginning of the algorithm.

In order to determine a lower bound for our algorithm, it is necessary to consider the worst case. In the worst case, C_1' is not an element of any optimal solution set; therefore, the insertion of C_1' keeps the algorithm from inserting some elements into the greedy solution set. Furthermore, after this first insertion, the maximum cardinality of the greedy solution set becomes $C – m_1 + 1$. Since m is the intersection number of sets from the optimal solution set. Also, we know that $C – m_1$ is greater or equal to 0 because negative cardinality is illogical. This makes the minimum value of result 1.

On the other hand, it is hard to define an upper bound for this algorithm also because it is possible that the first R elements of C' create one of the optimal solutions set regardless of the number of elements in C_i' and $|C'|$. As a result, the upper bound for the greedy solution set is R as we expected because the greedy algorithm solves correctly in some cases.

Thus, $\frac{R}{R} = 1 \geq \frac{GreedySolution}{OptimalSolution} \geq \frac{1}{R}$.

3. Algorithm Analysis

Brute Force Algorithm

A. Correctness Analysis

Our Claim: The algorithm we designed can find the largest cardinality subset D of a collection C of finite sets given as an input such that there will not be any common elements for all set pairs in D.

According to our algorithm, we are examining whether sets, which are in a chosen subset with length of L of C are disjoint or not. L starts from n to 1; therefore, if we manage to a subset with the length of L=n such that there will not be any common elements for all set pairs in this subset. The answer is n. Otherwise, the value of L decreases by 1 and the checking procedure starts again. This strategy goes until L=1 if the program can not find any subset D that satisfies the condition and the answer becomes 1.

We can prove our claim by using induction.

Proof:

(i) Basis Step: The case of N=1,

The maximum subset length of C is 1, and since there is no two set inside C that we can not choose to compare, the only set inside C holds the condition automatically. So, the answer is 1.

(ii) Inductive Step:

Let assume that when the size of our collection C is n, D' is a set consists of the largest cardinality subset D's; therefore, $D' = \{D_1, D_2, \ldots, D_y\}$ where $|D_i| = k$ for $i = 1, 2, \ldots y$. Now, we need to show that when the size of our collection C becomes n+1 by adding a new set into C, the algorithm finds the correct subset and returns either k or k+1 as the solution.

We know that the largest cardinality found is k before adding n+1th set; therefore, it is evident that there is no way to find a subset D with the cardinality of k+2 that holds the condition by adding only one set. Therefore, our program should compute and flow at least until L = k+1. Then, we need to make a comparison of each element of D_i's with the added set. If any D_i exists whose elements are disjoint with this newly added set, the program terminates, and the answer becomes k+1. Otherwise, L decreases by 1 and equals k. Then, the program must also terminate and give the output as k since the inductive assumption indicates that there are sets with the cardinality of k.

```
146
147 int main(int argc, const char * argv[]) {
148     srand(5);
149     int c_size = K-1;
150     vector < vector<int> > c (c_size);
151     vector<string> translated;
152     for(int i = 0; i < c.size(); i++) {
153         c[i] = randomVectorGenerator();
154         cout << "vector " << i + 1 << ":  ";
155         vectorPrinter(c[i]);
156     }
157     //cout << endl << "Subset With the Best Cardinality:\n";
158
159     vector<vector<string>> combtable;
160     combtable = findBitCombinations(K-1);
161
162     cout<<"finished"<<endl;
163     translated = vectorTranslator(c);
164     bool flag = false;
165     for(int i=K-1; i>0;i--){
166         for(int j=0;j<combtable[i].size();j++){
167             if(checkset(translated, combtable[i][j])){
168                 cout<<"Result: "<< i<<endl;
169                 flag =true;
170                 break;
171             }
172         }
173         if(flag)break;
174     }
175
176
177
178     return 0;
179 }
180
```

Figure1. The main function

B. Complexity Analysis

Since our algorithm starts after the line 157, we can divide it into the parts to calculate the complexity of this algorithm such as:

$$T(N = K - 1) = O(the\ lines\ between\ 159\ and\ 163) + O(the\ lines\ between\ 164\ and\ 174)$$

Between lines 159 and 163, there are two functions whose complexities are needed to be calculated and we can determine the complexity $O(1)$ for the other two lines easily:

$$O(the\ lines\ between\ 159\ and\ 163) = O(findBitCombinations(K - 1)) + O(vectorTranslator(c)) + O(1)$$

```
44
45  vector<vector<string>> findBitCombinations(int k)
46  {
47      string str = "";
48      string asd;
49      vector<string> dum;
50      vector<vector<string>> restable(k+1,dum);
51
52      // DP[k][0] will store all k-bit numbers
53      // with 0 bits set (All bits are 0's)
54      for (int len = 0; len <= k; len++)
55      {
56          DP[len][0].push_back(str);
57          str = str + "0";
58      }
59      // fill DP lookup table in bottom-up manner
60      // DP[k][n] will store all k-bit numbers
61      // with n-bits set
62      for (int len = 1; len <= k; len++)
63      {
64          for (int n = 1; n <= len; n++)
65          {
66              // prefix 0 to all combinations of length len-1
67              // with n ones
68              for (string str : DP[len - 1][n])
69                  DP[len][n].push_back("0" + str);
70
71              // prefix 1 to all combinations of length len-1
72              // with n-1 ones
73              for (string str : DP[len - 1][n - 1])
74                  DP[len][n].push_back("1" + str);
75          }
76          }
77
78      for(int com=1; com<=k;com++){
79          for(int m=0; m<DP[k][com].size();m++){
80              restable[com].push_back(DP[k][com][m]);
81          }
82      }
83      return restable;
84  }
85
```

Figure2. FindBitCombinations(int k)

In Figure2., the first lines are declarations of variables, which complexity is $O(1)$. Then, there comes a single for-loop that iterates k+1 times. In this loop, there is only one push_back operation and assignment operation. So, the complexity becomes $O((k+1)*2) = O(2k+2) = O(k)$.

Then, there comes another structure consisting of nested for-loops. In order to find the structure's complexity, we should investigate how many times the lines in the most-inner loops execute by considering the mechanism mentioned in Section 2. Since the push_back lines execute only once for each possible string consisting of 0's and 1's except the case of all 0's whose range is from 1 to k in length, the total executions are equal to $2^{k+1} - 11$. And there are a few executions belonging to lines 68 and 73, which are not counted, because they are executed but return false and they can not move into the loops. So, the number of total executions corresponds to $\frac{k(k+1)}{2} * 2 = k(k+1)$

($\frac{k(k+1)}{2}$ represents how many times the program enters into the first two loops and 2 represents inner two for-loops).

Consequently, the complexity of this structure is:

$$O(k(k+1) + 2^{k+1} - 11) = O(2^{k+1}) = O(2^k).$$

As explained in Section 2, the last nested for-loop transfers all the strings with the length of k we added into the matrix DP to the new variable 'restable', which means that the push_back operation executes $2^k - 1$ times here. It result in:

$$O(2^k - 1) = O(2^k).$$

Thus, $O(FindBitCombinations(K-1)) = O(1) + O(K-1) + O(2^K - 1) + O(2^K - 1) = O(2^K - 1).$

```
vector<string> vectorTranslator(vector<vector<int>> &myvec){
    int maxel=0;
    int minel=INT_MAX;
    int maxlen=0;
    int el;
    string temp;
    vector<string> translated;

    for(int i = 0; i < myvec.size(); i++) {
        for(int j=0; j < myvec[i].size();j++){
            el = myvec[i][j];
            if(maxel<el) maxel=el;
            if(minel>el) minel=el;
        }
        if(maxlen<myvec[i].size())maxlen=myvec[i].size();
    }
    string dumstr = "";
    for(int i=0; i<maxel;i++){
        dumstr += "0";
    }


    for(int i = 0; i < myvec.size(); i++) {
        temp = dumstr;
        for(int j=0; j < myvec[i].size();j++){
            el = myvec[i][j];
            temp[el-1]='1';
        }
        translated.push_back(temp);
    }
    return translated;
}
```

Figure3. vectorTraslator(vector<vector <int>> &myvec)

In Figure3., the first lines are declarations of variables, which complexity is $O(1)$. Then, the program flows to a nested for-loop with a few if statements and assignment operations that cost linear time. On the other hand, this nested loop is designed for finding max element, min element and max length by iterating all vectors inside “myvec”

and all elements inside those vectors. Therefore, its complexity becomes $O(myvec.size() * t)$ in the worst case where t represents the maximum size of vector in “myvec”.

In the following, a declaration and a loop within an assignment operation, in which the total complexity is $O(1) + O(t) = O(t)$.

At the end of the function, there is another nested loop iterating all vectors inside “myvec” and all elements inside those vectors in order to transform the subset in “myvec” and its elements into a string consisting of 0s and 1s. All lines inside this loop operate at complexity O(1), and since the loop structure iterates all elements inside the vectors in “myvec”, the total complexity equals $O(myvec.size() * t)$ in the worst case as above.

Thus, $O(vectorTranslator(c)) = O(1) + O(c.size() * t) + O(t) + O(c.size() * t) = O(c.size() * t) = O((K-1)t)$.

Since, $O(findBitCombinations(K-1))$ and $O(vectorTranslator(c))$ are known,

$O(the\ lines\ between\ 159\ and\ 163) = O(2^{K-1}) + O((K-1)t) + O(1) = O(2^{K-1})$ for so big K values.

Between lines 164 and 174, there is only one function whose complexity is needed to be calculated, and we can determine the complexity $O(1)$ for the other lines. Since the bounds of loops are known, we can easily express the complexity of the piece of code. For instance;

$\sum_{1}^{K-1} compatible[i].size() = \binom{K-1}{1} + \binom{K-1}{2} + ... + \binom{K-1}{K-2} + \binom{K-1}{K-1} = 2^{K-1} - 1$.

In the worst case, by determining the total number of executions line by line:

$O(the\ lines\ between\ 164\ and\ 174) = O(K-1) + O(2^{K-1} - K)$

$+(2^{K-1} - 1) * O(checkset(translated,\ comtable\,[i]\,[j])) + O(K-1) + 4 * O(1)$

$= O(2^{K-1}) + (2^{K-1} - 1) * O(checkset(translated, comtable[i][j]))$

where $O(K-1)$ for line 166, $O(2^{K-1} - K)$ for line 167, the other $O(K-1)$ for line 174 and $4 * O(1)$ for lines 169, 170, 171 and 174.

```
135
136 bool checkset(vector<string> vecs, string comb){
137
138     vector <string> chosen;
139     for(int i=0; i<K-1;i++){
140         if(comb[i]=='1'){
141             chosen.push_back(vecs[i]);
142         }
143     }
144     return isvalid(chosen);
145 }
146
```

Figure4. checkSet (vector<string> vecs, string comb)

In the worst case, comb equals to “11….11” with size is K-1. By determining the total number of executions line by line:

$O(checkset(vector < string > vecs, stringcomb))$
$= O(1) + O(K-1) + O(K-1) + O(K-1) + O(isvalid(chosen))$
$= O(K-1) + O(isvalid(chosen))$

```
bool isvalid(vector <string> chosen){
    int checkand=0;

    for(int i=0; i<chosen[0].size();i++){
        checkand=0;
        for(int j=0; j<chosen.size();j++){

            checkand += chosen[j][i] - '0';
        }
        if(checkand>1) return false;
    }
    return true;
}
```

Figure5. isvalid (vector<string> chosen)

In Figure5., the first line is a declaration of a variable, which complexity is $O(1)$. Then, the program flows to a nested for-loop with an if statement and a few assignment operations that cost linear time. On the other hand, this nested loop iterates chosen[0].size( )*chosen.size( ) in total. Before the function isvalid( ), we already set the size of strings inside of “chosen” to be the maximum number which is in one of the sets inside the given input set (This set corresponds to “c” in our C++ algorithms).

So, the complexity of this function is $O(m * chosen.size())$ where m = the maximum number which is in one of the sets inside the given input set.

Returning to the checkset(), we see that the number of 1s in “comb” determines the size of “chosen”. Therefore, the worst case of $O(isvalid(chosen))$ happens also when comb equals “11….11” with the size is K-1.

$$
\begin{aligned}
&O(checkset(vector < string > vecs, stringcomb)) \\
&= O(K-1) + O(isvalid(chosen)) \\
&= O(K-1) + O(m(K-1)) \\
&= O(m(K-1))
\end{aligned}
$$

Returning to the complexity of the lines between 164 and 174:

$O(the\ lines\ between\ 164\ and\ 174) = O(2^{K-1}) + (2^{K-1} - 1) *$
$O(checkset(translated, combtable[i][j]))$
$= O(2^{K-1}) + (2^{K-1} - 1) * O(m(K-1))$
$= O(2^{K-1}) + O(m * (K-1) * 2^{K-1})$
$= O(m * (K-1) * 2^{K-1})$

The final result is:

$T(N = K-1) = O(2^{K-1}) + O(m * (K-1) * 2^{K-1})$
$= O(m * (K-1) * 2^{K-1}))$
$= O(m * N * 2^N)$ where m represents the maximum number which is in one of the sets inside the given input set and N is the total number of sets in the given input C.

Greedy Algorithm

A. Correctness Analysis

Our Claim: The greedy algorithm we designed always works and gives a result, which is sometimes an optimal solution and sometimes not.

According to our algorithm, we need to sort the elements by their size from smallest to largest in the given set C at the first. Then, we define a solution set and insert the first element of the sorted set in it. Then the rest of the algorithm goes as follows:

Start to iterate the sorted set by starting from the second element to the last element. If the corresponding element and the elements in the solution set are disjoint, insert this element in the solution set. In the end, we have a solution set whose cardinality minimum 1 and maximum R which is the optimal solution.

We can show our claim as follows:

Except for the case when all sets in the given set C are disjoint, there is always a possibility that

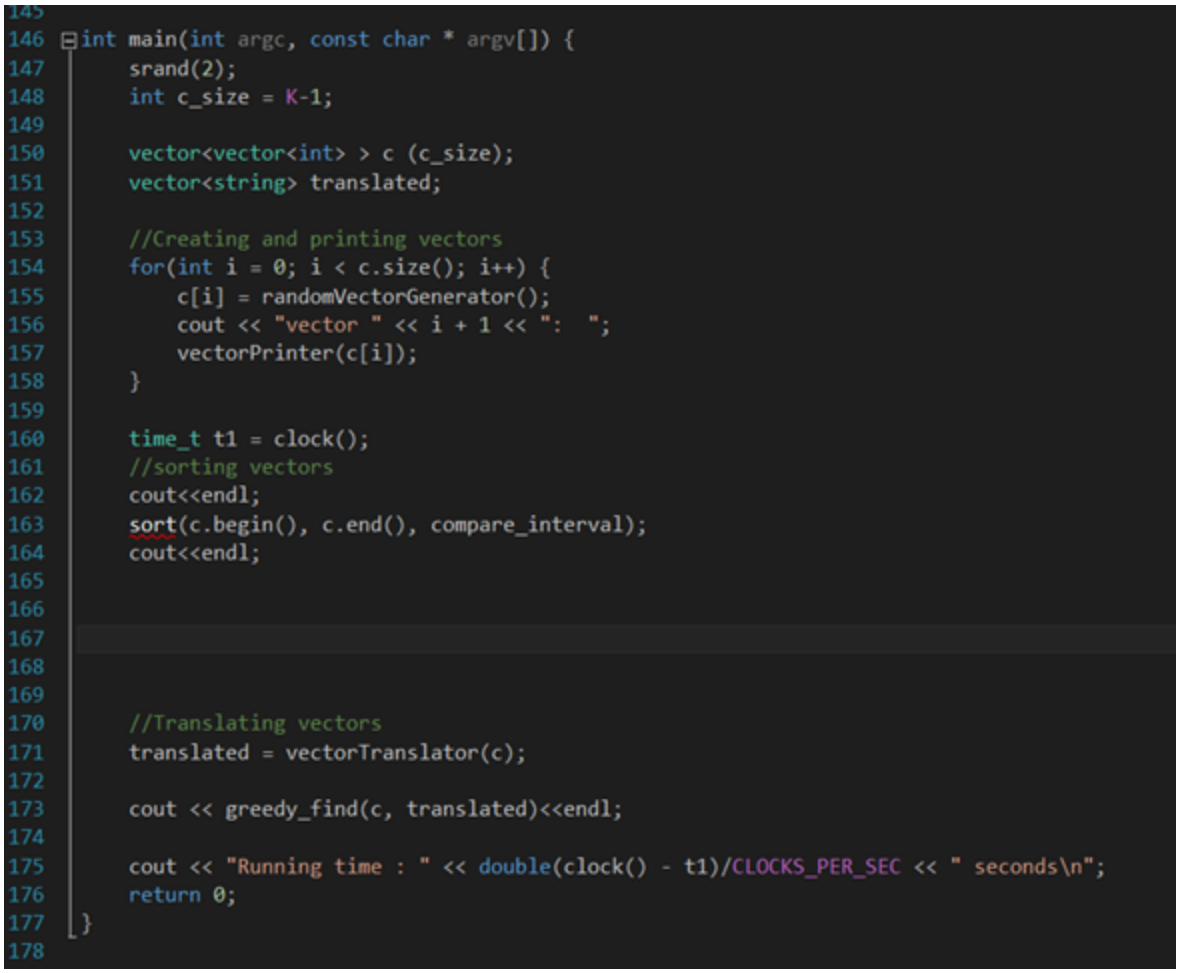
our algorithm’s output is 1 whatever the optimal solution is. In other words, let assume that the
first set in the sorted set has a different common element with the other sets. Thus, we reach 1
as the largest cardinality D of C that satisfies the disjoint condition.

On the other hand, the greedy algorithm’s output can equal the optimal solution. This equality holds if the following condition is satisfied by the sorted set:

- Let assume that the sorted set is $C' = \{C'1, C'2, \ldots Cn'\}$. The subset containing the first R
  set in C' equals one of the sets that give us the optimal solution (there can be many sets with the cardinality is K that satisfies the disjoint condition).

Then, our algorithm’s result will be R which is the optimal solution for our problem. For the other case, this algorithm will have resulted in an answer between 2 and $R–1$ after the iteration has been done.

B. Complexity Analysis



```
146 int main(int argc, const char * argv[]) {
147     srand(2);
148     int c_size = K-1;
149
150     vector<vector<int> > c (c_size);
151     vector<string> translated;
152
153     //Creating and printing vectors
154     for(int i = 0; i < c.size(); i++) {
155         c[i] = randomVectorGenerator();
156         cout << "vector " << i + 1 << ":  ";
157         vectorPrinter(c[i]);
158     }
159
160     time_t t1 = clock();
161     //sorting vectors
162     cout<<endl;
163     sort(c.begin(), c.end(), compare_interval);
164     cout<<endl;
165
166
167
168
169
170     //Translating vectors
171     translated = vectorTranslator(c);
172
173     cout << greedy_find(c, translated)<<endl;
174
175     cout << "Running time : " << double(clock() - t1)/CLOCKS_PER_SEC << " seconds\n";
176     return 0;
177 }
178
```

Figure1. The main function

If we ignore the parts of calculating time and creating our input parts, our algorithm is between lines 161 and 173. Between those lines, there are three functions that affect our complexity, **sort()**, **vectorTranslator()** and **greedy_find()**. So, the complexity of this algorithm is as follows:

$$T(N = K-1) = sort(c.begin(), c.end(), compare_interval) + O(vectorTranslator(c)) + O(greedy_find(c, translated))$$

Std::sort() is a generic function in C++ Standard Library, for doing comparison sorting and it is known that its time complexity of $NlogN = (K-1)log(K-1)$. From the brute-force analysis, it is known the complexity of **vectorTranslator(c)** as $O((K-1)t)$ where t represents the maximum size of a vector in c.

```
116
117
118 int greedy_find(vector<vector<int>> myvec, vector<string> translated){
119
120     vector<string> result;
121     vector<vector<int>> check_result;
122
123
124     if(translated.size() > 0){
125         result.push_back(translated[0]);
126         check_result.push_back(myvec[0]);
127     }
128     else return 0;
129
130     for(int i=1 ; i<translated.size() ; i++){
131         result.push_back(translated[i]);
132         check_result.push_back(myvec[i]);
133         if(!isvalid(result)){
134             result.pop_back();
135             check_result.pop_back();
136         }
137     }
138     return result.size();
139 }
140
```

Figure2. greedy_find(vector<vector <int>> &myvec, vector<string> translated

As you can see from Figure2., the complexity between lines 120 and 130 is $O(1)$. Then there comes a for-loop structure containing a function with the complexity of O(m(K-1)) at the worst case. The only thing we have to deal with is the translated.size() but we know that the string has been created by the maximum number which is in one of the sets inside the given input set. Thus,

O(greedy_find(c, translated)) = O(1) + O(m*m(K-1)) = O(m 2 (K-1)) where m = the maximum number which is in one of the sets inside the given input set.

Returning to the complexity of all algorithm, the final result is:
$O(N = K - 1) = sort(c.begin(), c.end(), compare_interval) + O(vectorTranslator(c)) + O(greedy_find(c, translated))$

$= O((K-1)log(K-1)) + O((K-1)t) + O(m^2(K-1))$

$= O((K-1) * max\{log(K-1), t, m^2\})$

$= O(N * max\{logN, t, m^2\})$

where t represents the maximum size of a vector in c and m represents the maximum number which is in one of the sets inside the given input set.

C. Space Complexity Analysis

The space complexity of the algorithm can be calculated for the same functions since the algorithm
is fully included in given lines and other parts are for sample generating. If we consider sort(),
vectorTranslator() and greedy_find() functions for space complexity then we can find following
results;
Sort() —> This is a generic function in C++ Standard Library and space complexity for this
the function is not defined but it can dynamically allocate memory. So we will take the worst case
for this one and it is n.

vectorTranslator() —> This function takes M**C where M is the maximum element in all sets and*
C is the number of sets.
Greedy_find() —> This function creates a new result list that may have the same size as the given set. Inside this function, there is no other memory is used except swap operations which
their space complexity is constant. So the space complexity for this function is C.
*Then we can conclude a total space complexity of M**C + M which is M*C since C is bigger than
1 and will dominate over M.

4. An Initial Testing of Implementation of the Algorithm

The algorithm for generating cases was designed to generate random sets as it shown in the code with parameters for maxSetSize, maxSetCount, and range of the elements.

```
vector<vector<int>> createSampleNorand(int subsetSize, int subsetCount, int max_elem)
{
    vector<vector<int>> finalVector;//initalize the set that will be returned

    for (int i = 0; i < subsetCount; i++)
    {
        vector<int> tempSet;

        for (int a = 0; a < subsetSize; a++)
        {
            int randNum = rand() % max_elem + 1;
            tempSet.push_back(randNum);
        }
        finalVector.push_back(tempSet);
    }

    return finalVector;
}
```

This code is generating sample case for the algorithm and you can see the results of test cases when the cardinality of input set is 5, 8, 12 and 15

1. **For cardinality 5**

```
vector 1:  45  24  46
vector 2:  13  20  49  24  7  3  31  22
vector 3:  35  21
vector 4:  45  3  10
vector 5:  4  1  47  12  21
finished
Result: 2
Running time : 5.6e-05 seconds
```

2. **For cardinality 8**

```
vector 1:  45  24  46
vector 2:  13  20  49  24  7  3  31  22
vector 3:  35  21
vector 4:  45  3  10
vector 5:  4  1  47  12  21
vector 6:  49  36  34
vector 7:
vector 8:  13  35  9
finished
Result: 5
Running time : 0.000153 seconds
```

3. **For cardinality 12**

```
vector 1:  45  24  46
vector 2:  13  20  49  24  7  3  31  22
vector 3:  35  21
vector 4:  45  3  10
vector 5:  4  1  47  12  21
vector 6:  49  36  34
vector 7:
vector 8:  13  35  9
vector 9:  4  4
vector 10:  8  38  22  24
vector 11:  34  25  49  17  23  41  14  21  43  23  43
vector 12:  26  42
finished
Result: 7
Running time : 0.003415 seconds
```

4. **For cardinality 15**

```
vector 1:  45  24  46
vector 2:  13  20  49  24  7  3  31  22
vector 3:  35  21
vector 4:  45  3  10
vector 5:  4  1  47  12  21
vector 6:  49  36  34
vector 7:
vector 8:  13  35  9
vector 9:  4  4
vector 10:  8  38  22  24
vector 11:  34  25  49  17  23  41  14  21  43  23  43
vector 12:  26  42
vector 13:  15  22  36  1  23  17  21  38  40  2
vector 14:  46  8  28  27  31  26  27  30  34  4
vector 15:  20  45
finished
Result: 7
Running time : 0.076899 seconds
```

5. Experimental Analysis of the Performance (Performance Testing)

We have used the “getRunningTime” function for the performance testing. This function uses mea time, standard deviation, standard error and t values based on confidence values, which we used %90 and %95.

```
void getRunningTime(vector<double> runningTimes)
{
    double totalTime = 0.0;
    int N = runningTimes.size();
    for (int i = 0; i < N; i++)
    {
        totalTime += runningTimes[i];
    }

    double standardDeviation = calculateSD(runningTimes);

    double m = totalTime / N;

    //const double tval90 = 1.660;
    //const double tval95 = 1.904;
    const double tval90 = 1.676;//N=50
    const double tval95 = 2.009;

    double sm = calculateStandardError(standardDeviation, N);

    double upperMean90 = m + tval90 * sm;
    double lowerMean90 = m - tval90 * sm;

    double upperMean95 = m + tval95 * sm;
    double lowerMean95 = m - tval95 * sm;

    cout << "Mean Time: " << m << " ms" << endl;
    cout << "SD: " << standardDeviation << endl;
    cout << "Standard Error: " << sm << endl;
    cout << "%90-CL: " << upperMean90 << " - " << lowerMean90 << endl;
    cout << "%95-CL: " << upperMean95 << " - " << lowerMean95 << endl;
    runningTimes.clear();
}
```

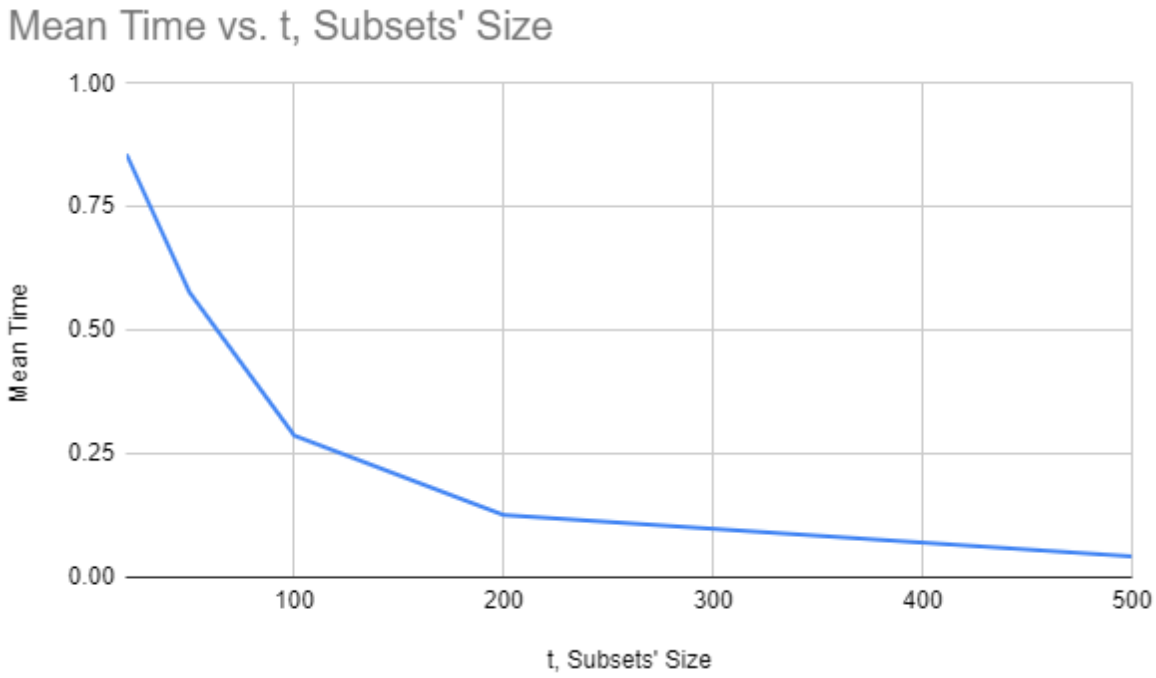
Visualization

In our algorithm the values which affect the complexity is C's size and t which is each subset's size. Therefore we will be experimenting with changing these values:

A) In here we have kept the C's size constant (20) and changed t.

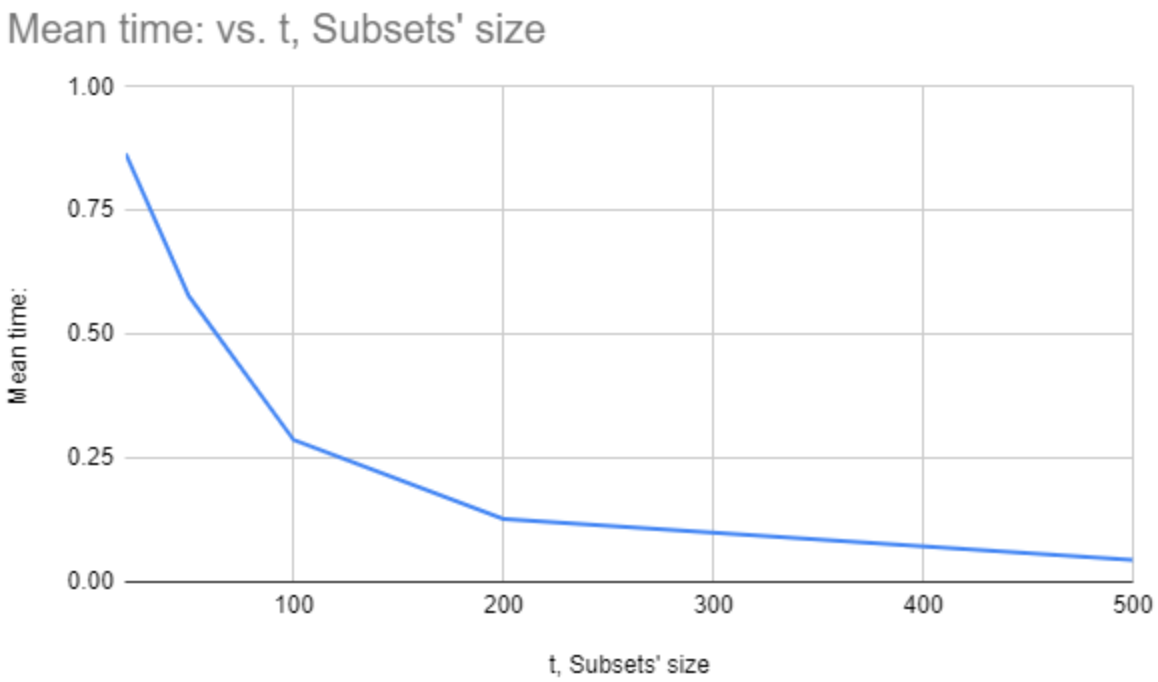
t, Subsets' size	Mean time:	Standard Deviation:	Standard Error:	%90 CL	%95 CL
20	0.85688 ms	0.0772224	0.0109209	0.875009 - 0.838751	0.877673 - 0.836087
50	0.57728 ms	0.0737455	0.0104292	0.594592 - 0.559968	0.597137 - 0.557423
100	0.28686 ms	0.0413251	0.00584425	0.296561 - 0.277159	0.297987 - 0.275733
200	0.12648 ms	0.0196969	0.00278557	0.131104 - 0.121856	0.131784 - 0.121176
500	0.04258 ms	0.00816845	0.00115519	0.0444976 - 0.0406624	0.0447795 - 0.0403805

a)1) This table shows the statistics when N is 50, the algorithm is repeated 50 times.



t, Subsets' size	Mean time:	Standard Deviation:	Standard Error:	%90 CL	%95 CL
20	0.86478 ms	0.0824418	0.00824418	0.878597 - 0.850963	0.881343 - 0.848217
50	0.57788 ms	0.0930742	0.00930742	0.593479 - 0.562281	0.596579 - 0.559181
100	0.28661 ms	0.0500418	0.00500418	0.294997 - 0.278223	0.296663 - 0.276557
200	0.12728 ms	0.0190552	0.00190552	0.130474 - 0.124086	0.131108 - 0.123452
500	0.04378 ms	0.00797193	0.000797193	0.0451161 - 0.0424439	0.0453816 - 0.0421784

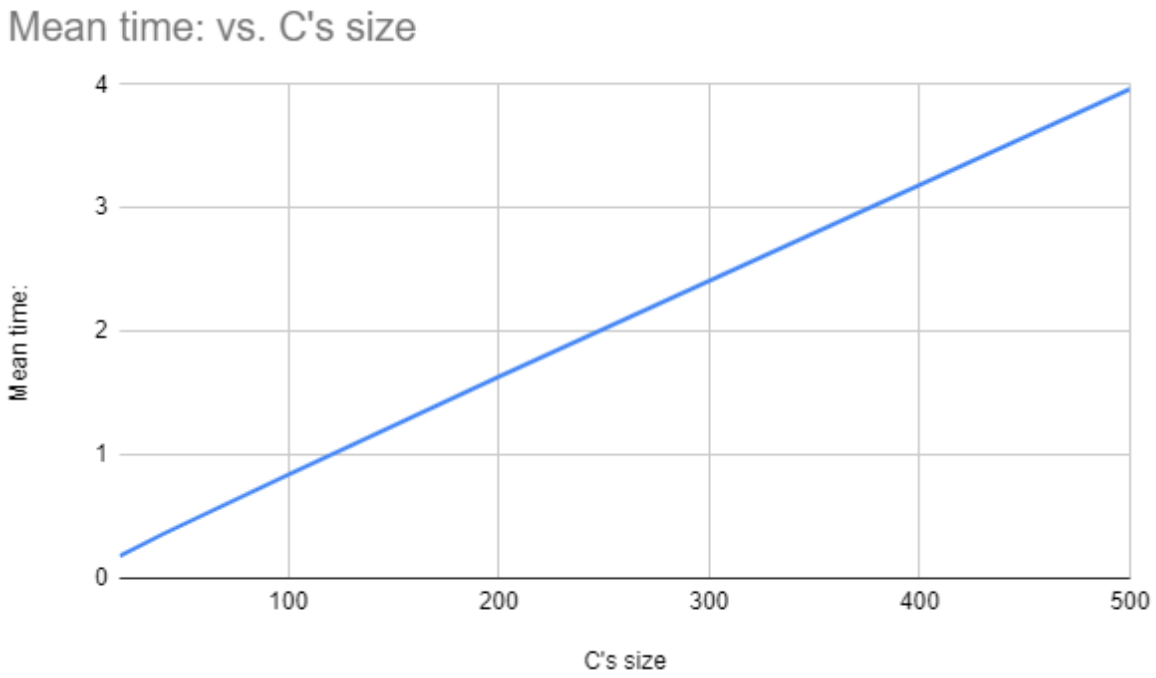
a)2) This table shows the statistics when N is 100, the algorithm is repeated 100 times.



B) In here we have kept the subset's sizes constant (150) and changed C's cardinality.

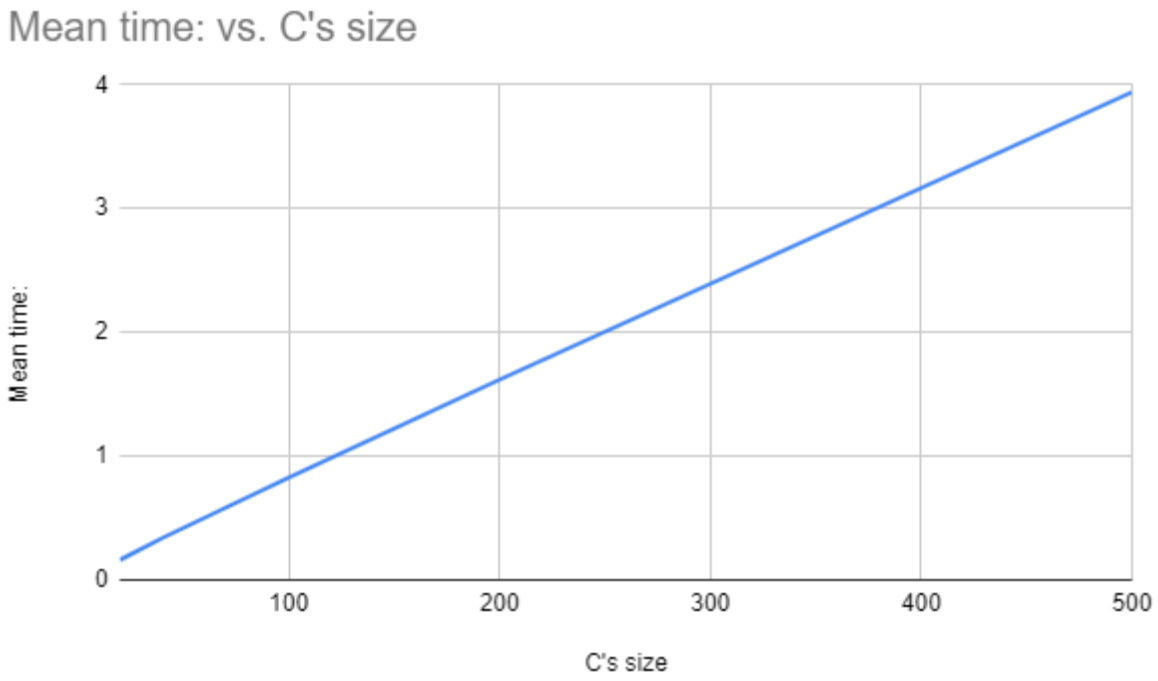
C's size	Mean time:	Standard Deviation:	Standard Error:	%90 CL	%95 CL
20	0.17948 ms	0.0292152	0.00413166	0.186405 - 0.172555	0.18778 - 0.17118
40	0.35152 ms	0.0451213	0.00638111	0.362215 - 0.340825	0.36434 - 0.3387
100	0.83852 ms	0.0620201	0.00877096	0.85322 - 0.82382	0.856141 - 0.820899
200	1.63092 ms	0.112711	0.0159398	1.65764 - 1.6042	1.66294 - 1.5989
500	3.96198 ms	0.209957	0.0296923	4.01174 - 3.91222	4.02163 - 3.90233

b)1) This table shows the statistics when N is 50, the algorithm is repeated 50 time



C's size	Mean time:	Standard Deviation:	Standard Error:	%90 CL	%95 CL
20	0.16063 ms	0.0024098	0.0024098	0.164724 - 0.156536	0.165558 - 0.155702
40	0.33498 ms	0.0391809	0.00391809	0.341637 - 0.328323	0.342992 - 0.326968
100	0.82617 ms	0.0785143	0.00785143	0.83951 - 0.81283	0.842226 - 0.810114
200	1.61662 ms	0.108847	0.0108847	1.63511 - 1.59813	1.63888 - 1.59436
500	3.93867 ms	0.216175	0.0216175	3.97456 - 3.90278	3.97983 - 3.89751

b)2) This table shows the statistics when N is 100, the algorithm is repeated 100 times.



Brute Force Running Time Comparison

We have compared how both algorithms behave when their N (number of repeats) and C's size changes. However brute force algorithm took too long in some of the test cases and failed therefore those are left empty.

no Repeats / C's Size	5	10	20
5	0.0223880597	0.0078125	0.00028223696
10	0.0001076403		
15	0.00071542797		

6. Experimental Analysis of the Correctness (Functional Testing)

Black Box Testing.

1. C = ∅ , for empty set

2. C = {{1}, {2}, {3}, ... {N} }, for sets with only 1 element
3. C = {{1,2,3},{4,5,6},{7,8,9}}
4. C = {{1,2,3,4}, {2,32,12}, {3,43,5}, {4,63,213}}

	Experimental Results	Expected Results
1	0	0
2	N	N
3	3	3
4	1	1

As expected the implemented code is working correctly for trial edge cases.

White Box Testing

Our code does not have many if conditions other than some checks which are used for checking whether our input set is empty or not, thus we have a few edge cases that we need to check for full statement coverage. One is for checking if our input is an empty set and another edge case is when we have disjoint sets as input.
Edge Cases:

1) C = $\emptyset$, for empty set coverage

2) C = { $\emptyset$, $\emptyset$,$\emptyset$,$\emptyset$ } for having multiple empty set

3) C = {{1,2,3,4,5},{6,7,8,9},{10,11,12},{14,15,32}}, for disjoint set coverage.
In this case the code never runs on some lines of codes where it checks the validity of the sets.

4) Another edge case is that if there is no disjoint sets, the code still picks a set as an answer.
C = {{1,2,3},{2,4,5},{3,6,7}}, it chooses the answer as 1 (first set).

5) C = {{1,2,3,4,5},{6,7,9},{10,11},{13}} for the order of elements goes largest to smallest while the all elements are disjoint.

6) C = {{1,2,3,4,5},{6,7,8,9},{9,10,11},{10,13}} for the order of elements goes largest to smallest while some elements are not disjoint.

	Experimental Result	The Expected Resut
C = { ∅ }	0	0
C = { ∅ , ∅ ,∅ ,∅ }	1	1
C = {{1,2,3,4,5},{6,7,8,9},{10,11,12},{14,15,32}}	4	4
C = {{1,2,3},{2,4,5},{3,6,7}}	1	0
C = {{1,2,3,4,5},{6,7,9},{10,11},{13}}	4	4
C = {{1,2,3,4,5},{6,7,8,9},{9,10,11},{10,13}}	3	3

7. Discussion

The results of both algorithms have their own advantages and disadvantages. For example, brute force code works perfectly and gives the optimum solution but the time and space complexity of the code is exponential so it is not feasible to use it for big input sets like N > 25. Besides, that Greedy Algorithm is pretty fast even for 1000 instances but it does not guarantee the best result and also gives poor results in some special cases.

In the experimental performance analysis part results have surprised us since the runtime is decreased while we increased the length of sets which is indicated with t in part A. This surprised us but then we realized that the result is also decreasing when we increase t. This is because it is more likely to have an intersection when there is more element in each set. So in this case the run time is decreasing because the result is decreasing. The reason for this situation is the complexity of the isvalid() function. It depends on the number of input sets and the range of those sets. In our experiment range is constant but the number of inputs is decreasing while we increase t. It makes sense for this function to affect overall running time by nearly x10 because the complexity of the algorithm is dominated by isvalid() function.

8. References

Richard M. Karp (1972). "Reducibility Among Combinatorial Problems" (PDF). In R. E. Miller; J. W. Thatcher; J.D. Bohlinger (eds.). *Complexity of Computer Computations*. New York: Plenum. pp. 85–103. doi:10.1007/978-1-4684-2001-2_9. ISBN 978-1-4684-2003-6.

Schreiber, J. & Migler, T. (2021, March 3). Original complexity reductions from set packing to clique and from hitting set to satisfiability [Conference session]. 2021 Computer Science Conference for CSU Undergraduates, Virtual. httpsJ/cscsu-co nf erence. g ith u.b. io/index. html