1 Problem 1

Use solution provided in Exam Period

2 Problem 2

2.1 Mathematical Formulation

Given a dictionary of size D with words $d_i, i \in [1, D]$ of average length l construct a Ternary Search Trie based algorithm that, when given k queries, q_i, of average length n, will retrun all strings in the dictionary such that $q_i \in /d_j/$.

2.2 Solution

Important Confusing Data Structures:

- Trie<String>, HashSet<String>>**T** : Contains the suffixes of each word d_i in the dictionary of length 1..—d_i— with HashSets associated with each letter

The main functionality of this will be to build up a suffix Ternary Search Trie with each node refering to a letter containing a hashset of the words that have the string thusfar to get to this node as a subset.

Algorithm 1 Main

```
procedure BUILD(StringSet dictionary)
    T ← initialized
    for word d_i in dictionary do
        for s ∈ [0, (|d_i| − 1)] do
            sub ← d_i.SUBSTRING(s,|d_i| − 1)
            T.INSERT(sub, d_i) // see below
procedure INSERT(String key, String word)
    for each letter in key as we traverse through Nodes do
        if Node exists then
            if
                T.GET(letter).CONTAINS(word) then return;
            else
                T.GET(letter).PUT(word)
        else
            HashSet<String> hs ← initialize, put in word
            T.ADD(letter, hs)
procedure SEARCH(String query)
    HashSet<String> hs ← T.GET(query)
    for String s ∈ hs do
        PRINT(s)
```

2.3 Correctness

Proposition 1.

This construction will enable us to determine all strings in our dictionary which contain the substring of query.

Proof.

As we build up our Trie, we maintain the order of letters in each word d_i from the dictionary. We also insert every suffix of the word from length $|d_i| \to 1$ so that way every single letter contained is accessible, and every pair, and every triple,... Since every time we insert a portion of each suffix we are inserting the original word into a stored hashset at each node, we simply will have to search for the query and we will get all of the words (with no repeats since it is a hashset) that contain that query as a substring. □

2.4 Analysis

For the following analysis, we will say that D is the size of our dictionary l is the average length of our dictionary words, m is the average number of words returned

per qeury search, n is the average length of our query words and k is the number of querys.

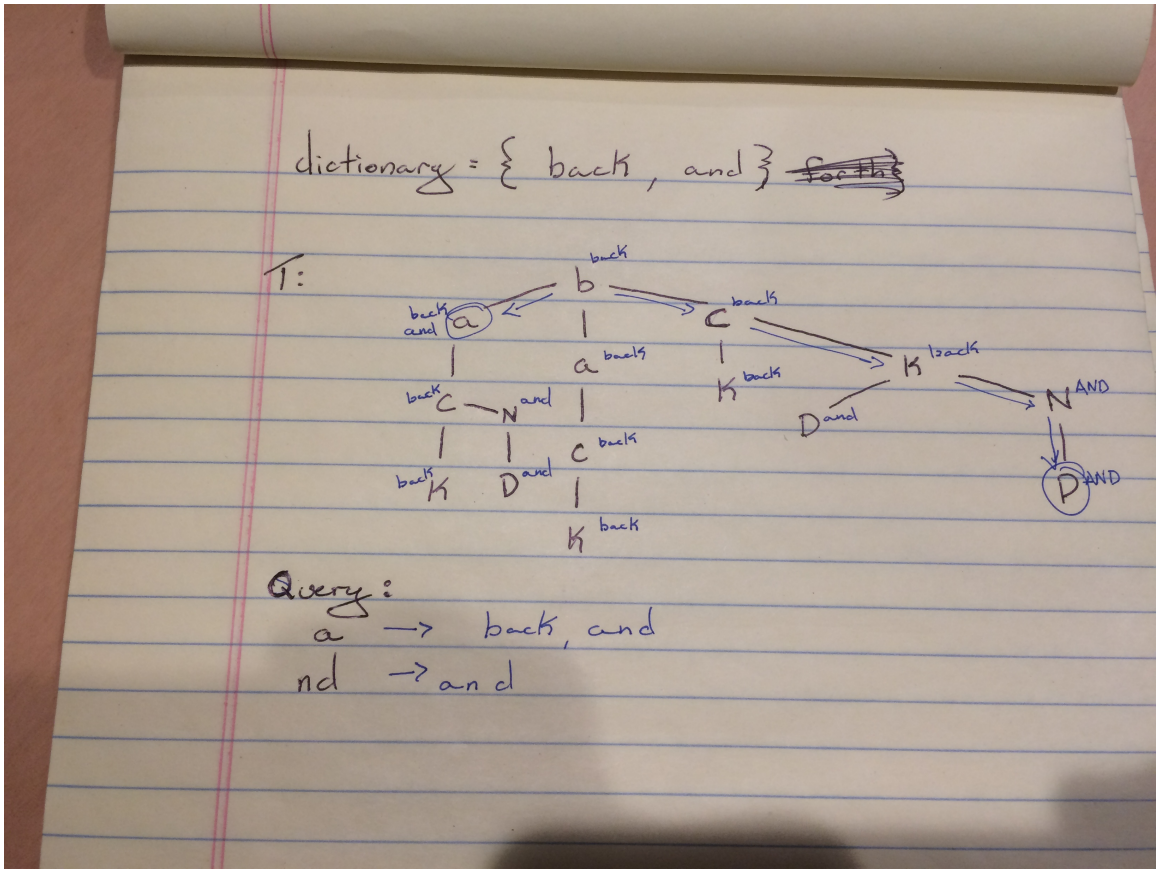
Proposition 2. *The time complexity of this algorithm is* $\boldsymbol{O(D \cdot l^2 + k \cdot (n+m))}$

Proof. We will split this into two processes, building the Trie and returning all words with the querys. For building the Trie we go through every word in the dictionary O(D) and for each one we compute all of the suffixes of length $1..l$ $O(l^2)$ which then each get put into the Trie such that each letter will enter the original word d_i into a hashset $O(1 \cdot l) \implies D \cdot (l^2 + l) = D \cdot l^2$. For the query aspect then we have, for each query O(k), we traverse our Trie searching for the query O(n), and print out each word stored in the corresponding hashset.

Giving us a time complexity of $\mathbf{O(D \cdot l^2 + k \cdot (n+m))}$

□

2.5 An Example



3 Problem 3

3.1 Mathematical Formulation

Given a network of size N which forms a rooted tree if the sink and all its incident edges are removed, give a worst case linear time algorithm to determine the maxflow for the set.

3.2 Solution

Important Confusing Data Structures: (we asume that these are filled out already)

- LinkedList[N] **edgeOut** : stores the indices of the nodes that the
- int[N][N] **cap** : keeps track of capacities between node i and node j
- int[N] **parent** : keeps track of who the parent is for each node i

The main functionality of this will be to recursively determine the maximum capacity that each edge has with the base case being node connected to the sink and the sum of the flow values from the source to the nodes that it connects to will be the value we return.

Algorithm 2 Main

```
procedure CONTROL
    flowValue = 0
    for each node n in edgeOut[0] do
        flowValue += CALCFLOW(node)
    PRINT(flowValue)
procedure CALCFLOW(int node)
    value = 0; p = parent[node]
    if node is sink then
        return cap[p][sink]
    else
        for node n ∈ edgeOut[node] do
            value += calcFlow(node)
    return MIN(cap[p][node], value)
```

3.3 Correctness

Proposition 3.
This implementation will give us the correct max flow value for a graph which forms a rooted tree if the sink and all its incident edges are removed

Proof.
This is due to the properties of the combination of max flow and a rooted tree. With a rooted tree, each node has 1 parent. This is useful because we then have a basis for the the maximum flow that can come into the node, then we can calculate the maximum flow that can go out of a node. While doing this recursively we simply use:

$$maxFlow(node) = \begin{cases} capacity(inEdge)\ if\ node = sink \\ minimum \begin{cases} capacity(inEdge) \\ \sum_i flow_i FromNode \end{cases} , else \end{cases}$$

Therefore our base case is the sink and we build backwards from there taking the minimum of the capacity of the edge leading into the node and the sum of the flows of the forward edges.

When we have recursively called this, each of the flowValues going from source to each node it connects to has the appropriate value of the maximum flow that can be pushed through that node. Therefore summing them up will give us the maximum flow for the graph □

3.4 Analysis

For the following analysis, we assume that we have N nodes and at most 2·N edges.

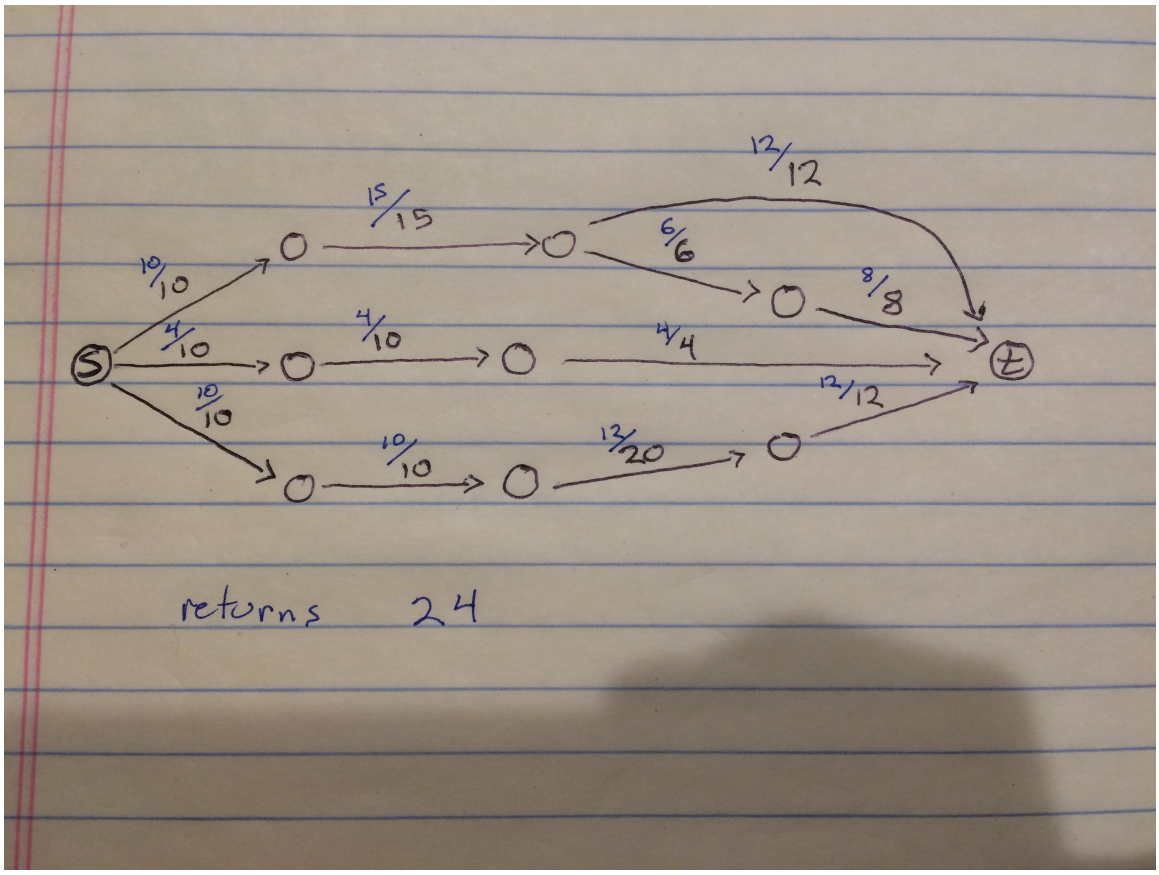
Proposition 4. *The time complexity of this algorithm is* ***O(N)***

Proof. This is due to the fact that we will essentially only visit each node 1 time since each node can only have one parent (due to the tree property) $\implies$ we have a linear algorithm.

Giving us a time complexity of **O(N)**

□

3.5 An Example



4 Problem 4

4.1 Mathematical Formulation

Given two sets d_1, d_2 which are both of size N and are sorted in ascending order, determine, when combined, what the median number is between the two of them i.e. the n^{th} smallest number.

4.2 Solution

The main functionality of this algorithm is to perform a form of set reducing keeping with the knowledge that half of the numbers are smaller and half of the numbers are bigger every time from each sub set recurance happening. Therefore when we get down to just two numbers we can just pick the smaller. So to start off this algorithm we would use $System.out.println(calculateMid(d_1, d_2, 0, N-1, 0, N-1))$

Algorithm 3 Main

procedure CALCULATEMID($d_1, d_2, l_1, h_1, l_2, h_2$)
  $mid_1 = l_1 + \lfloor \frac{h_1 - l_1}{2} \rfloor$; $mid_2 = l_2 + \lfloor \frac{h_2 - l_2}{2} \rfloor$;
  $m_1 = d_1[mid_1]$; $m_2 = d_2[mid_2]$;
  if $m_1 > m_2$ **then**
    if $l_1 == m_1 \&\& l_2 == m_2$ **then**
      return m_2
    $h_1 = mid_1, l_2 = mid_2, diff_1 = h_1 - l_1, diff_2 = h_2 - l_2$
    if $diff_1 == diff_2$ **then**
      return CALCULATEMID(l_1, h_1, l_2, h_2)
    else if $diff_1 > diff_2$ **then**
      return CALCULATEMID($l_1, h_1 - 1, l_2, h_2$)
    else
      return CALCULATEMID($l_1, h_1, l_2 + 1, h_2$)
  else
    if $l_1 == h_1 \&\& l_2 == h_2$ **then**
      return m_1
    $l_1 = mid1, h_2 = mid2, diff_1 = h_1 - l_1, diff_2 = h_2 - l_2$
    if $diff_1 == diff_2$ **then**
      return CALCULATEMID(l_1, h_1, l_2, h_2)
    else if $diff_1 > diff_2$ **then**
      return CALCULATEMID($l_1 + 1, h_1, l_2, h_2$)
    else
      return CALCULATEMID($l_1, h_1, l_2, h_2 - 1$)

4.3 Correctness

Proposition 5.
This algorithm will correctly give us the n^{th} lowest number given an input of two distinct sets d_i, d_j of ordered integers each of length N.

Proof.
Mathematically speaking each time that we perform this algorithm, we are decreasing our search set size by half or half - 1 depeneding on the initial size i.e. if initial size of the set is N and N/2 is a fraction and not a whole number, then we will be reducing by N/2 - 1 everytime. Regardless we keep reducing and everytime we are gauranteed that a quarter of our set is smaller and a quarter of our set is larger than some range.

This is because when we reset our pointers everytime we are doing so with the intention that if $m_1 \in d_1 < m_2 \in d_2$ Then we can be assured that all numbers less than m_1 are not our midpoint and all numbers greater than m_2 are not are midpoint. Therefore we continue like this until we are left with only one number in d_1 and one number in d_2 at which point we know that there are $N-1$ numbers less than them and $N-1$ numbers greater than them. Therefore choosing the smaller of the two will give us the N^{th} smallest. □

4.4 Analysis

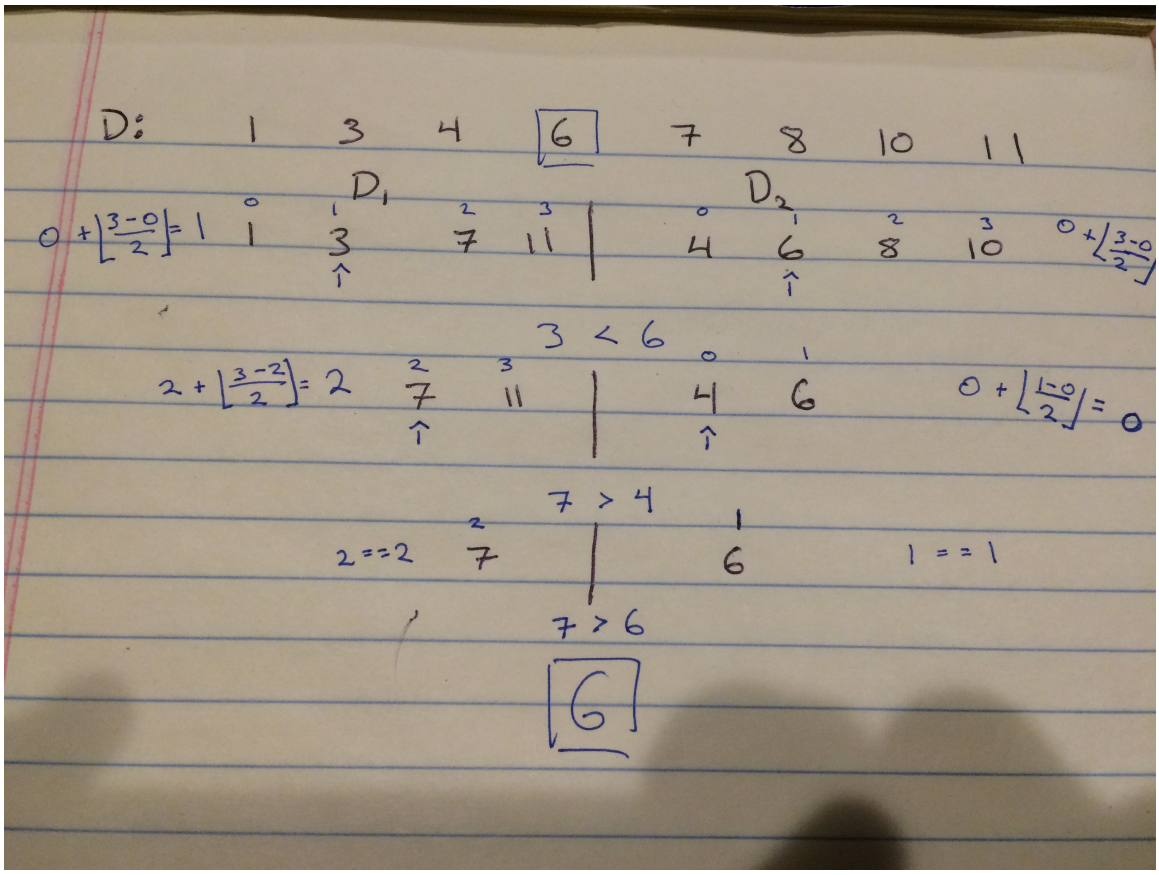
Proposition 6. *The time complexity of this algorithm is* ***O(log(N))***

Proof. According to the Masters Theorem since our ruccurence is $T(N) = T(N/2)+2$, this is a log(N) problem

Giving us a time complexity of **O(log(N))**

□

4.5 An Example



5 Problem 5

This problem is NP-Complete.

We begin by prooving NP by describing the certificate in polynomial time meaning, given k sets of jobs, say we have h hours in total, for each job, we check to see that every other job does not have any overlapping hours. This can be done in h^2 time by simply checking for each hour if there exists another hour in another job that overlaps it.

Next we must proove that I is NP-Hard. We do this by polonomially reducing it from an Independent Set i.e. let this problem be I, $INDEPENDENT\_SET \leq_P I$. Given a graph, if we can find an independent set of at least size k then we are gauranteed to be able to find a set of jobs of at least size k. We do this by having each vertex $v \in G$ represent a job $j \in I$ and each edge, $e_{l,k} = v_l - v_k$, represent an overlapping time interval between jobs j_l and j_k. Therefore we can say that $\exists$ a set of nodes $v_i \in G$ which are an independent set of size $\geq k$ iff $\exists$ a set of jobs $j_i \in I$ which do not overlap of size $\geq k$.

($\Rightarrow$) Given an independent set of at least size k then we know that there are at least k nodes with no edges between them, therefore if we convert to a set of jobs as described above, there would exist at least k jobs with no overlapping times which solves our problem.

($\Leftarrow$) Given a set of at least k jobs with no overlapping times, then we can see that if each job where a node and each overlapping time an edge between two of those nodes, that there would be an independent set of at least size k as well.

Therefore since both NP and NP-Hard, we know that this problem is NP-Complete. □

6 Problem 6

6.1 Mathematical Formulation

Given N Districts numbered 1..N ($\{P_1...P_N\}$), each of which having m registered voters makine $N \cdot m$ registered voters in total which each votes for either party A or party B. We will determine wheather or not it is possible for either party to be GerryMandered

6.2 Solution

The main functionality of this algorithm is to treat it as a DP problem. The first thing that we do is separate the sets of A-majority districts and B majority districts keeping track of what the running difference is in each group. If the difference is equal, then we know that this is not susceptible to gerrymandering (see corallary 7). Otherwise we choose whichever one has a larger difference as what we should determine to be true.

Algorithm 4 Initialization and Control Method

```
procedure CONTROL(set P)
    A, B ← initialized; sumA, sumB = 0; N = size(P)
    for p_i ∈ P do
        diff = p_i.A − p_i.B
        if diff ≥ 0 then
            A.insert(diff)
            sumA += diff
        else
            diff *= -1
            B.insert(diff)
            sumB += diff
    if sumA > sumB then
        if size(A) == 1 then return false;
        return hasGM(A, sumA, B, sumB, N)
    else if sumA < sumB then
        if size(B) == 1 then return false;
        return hasGM(B, sumB, A, sumA, N)
    else if sumA == sumB then
        return false;
```

Once we have decided which set is a viable candidate for gerrymandering, W.L.O.G. we can denote this as set_1 which has a majority of value sum_1 and the one which is not as set_2 with a majority of value sum_2. What we do then is arrange 2 sets of

differences for those with set_1 as the dominant and another for those with set_2 such that they are organized by number of Districts that comprise them. i.e. level 2 has all combinations of size 2 of districs. (See proof for more details)

Algorithm 5 Dynamic Programming Solution

```
procedure hasGM(set_1, sum_1, set_2, sum_2, N)
    LinkedList<Integer>[ ] S_1 ← buildDpTable(set_1, sum_1, N)
    LinkedList<Integer>[ ] S_2 ← buildDpTable(set_2, sum_2, N)
    absoluteDiff ← sum_1 − sum_2
    for numElmt_1 ∈ [S_1.length − 1, 0] do
        numElmt_2 ← (N/2 − numElmt_1) − 1
        if numElmt_2 == −1 then
        else
            for int e_1 ∈ S_1[numElmt_1] do
                for int e_2 ∈ S_2[numElmt_2] do
                    diff ← e_1 − e_2
                    if (e_1 − e_2 ≤ 0) || (e_1 − e_2 ≥ absoluteDiff) then
                        continue;
                    return true;
    return false;
```

Algorithm 6 Helpers

```
procedure buildDpTable(set, size, N)
    N = minimum(set.size(), N/2); height = 0;
    boolean[N][size]dp ← initialized
    for district, d ∈ set do
        for row ∈ [height, −1] do // decrease
            if row == -1 then
                dp[0][d] = true;
            else
                for i ∈ [0, size − 1] do
                    if dp[row][i] && (i + d) < size then
                        dp[row+1][i + d] = true;
            if height < N-2 then height++
    return getBag(dp);
procedure getBag(boolean[][] set_dp)
    LinkedList<Integer >[set_dp.length] bag ← initialized
    for row ∈ set_dp.length do
        for col ∈ set_dp[row].length do
            if set_dp then
                bag[row].insert(col)
    return bag;
```

6.3 Correctness

Corollary 7.

Given N Precincts $= P$, if we define the total number of voters for party A as $sumA = \sum_{i=1}^{N}(A \text{ voters in } p_i)$ and likewise the total number of voters for party B as $sumB = \sum_{i=1}^{N}(B \text{ voters in } p_i)$ then the difference, $diff = sumA - sumB$. if $diff \leq 0$ then it is impossible for Gerrymandering is impossible.

Proof.

We will proove by contradiction. Say Gerrymandering is possible for party A, then that implies that we can have two subsets P_1' and P_2' s.t. $P_1' \cup P_2' = P$ and $P_1' \cap P_2' = \emptyset$ and the sum of voters for A in $P'1$ ($sumA_1$) > the sum of voters for B in $P'1$ ($sumB_1$) and same for those in P_2' ($sumA_2 > sumB_2$). This would mean then that $sumA_1 - sumB_1 > 0$ and $sumA_2 - sumB_2 > 0$ which means that $(sumA_1 - sumB_1) + (sumA_2 - sumB_2) > 0$. However $(sumA_1 - sumB_1) + (sumA_2 - sumB_2) = (sumA_1 + sumA_2) - (sumB_1 + sumB_2) = sumA - sumB \leq 0$ which gives us a contradiction. □

Proposition 8.

Given two sets of differences that are organized by number of precincts used to

create the returned value D_A, D_B (i.e. $D_A(3)$ contains a list of integers which have been composed using 3 districtcs) and the total difference between sets 1 and 2 being a positive number $diff_D$, we can determine wheather or not Gerrymandering is possible.

Proof.

We first define how $diff_D$ is involved. We know that $diff_D = SUM(A) - SUM(B)$ and this we can say is $sumA - sumB = (sumA_1 + sumA_2) - (sumB_1 + sumB_2) = (sumA_1 - sumB_1) + (sumA_2 - sumB_2)$ where $sumA_1$ corresponds to a subset of A-dominant districts and $sumB_1$ corresponds to a subset of A-dominant districts. Then it follows by Gerrymandering definition that for A to be succeptible, it must win in both sub-groups and the two subgroups must be of equal size. Therefore we can say that both $(sumA_1 - sumB_1)$ and $(sumA_2 - sumB_2)$ must be positive i.e. >0. Since we have the relation then that $(sumA_1 - sumB_1) + (sumA_2 - sumB_2) = diff_D \implies (sumA_1 - sumB_1), (sumA_2 - sumB_2) < diff_D$ since neither can be positive. Therefore it is sufficient to show that only one of these is within those bounds since it would gaurantee the other to be as well (similar argument as in Corollary above). So what we do then is to create sets using our given D_A, D_B of set sizes that are equal to $\frac{N}{2}$ and checking to see each combination if the difference $e_1 - e_2,\ where\ e_1 \in D_A[i], e_2 \in D_B[\frac{N}{2} - i]$ is within the bounds. When we have done this we are gauranteed that the complement set (not important what it is) is also within the bounds $\therefore$ Gerrymandering is possible. If we enumerate every possiblity and nothing is found, then we are guaranteed that one does not exist since we looked at every possibility. □

6.4 Analysis

Proposition 9. *The time complexity of this algorithm is* $\boldsymbol{O(N^2 \cdot (N + D))}$

Proof. This is because our initial control algorithm take only linear time to build up the two sets O(N), then we make two boolean tables, each one involving: for each element in the set O(N), for each of the N rows (worst case) O(N), we fully traverse the row O(D) $\implies$ $O(N^2 \cdot D)$. Then we create a bag for each one of the tables which takes linear time in relation to the table $O(N \cdot D)$. Once we have done this, for each row in bag_1 O(N) we will compare each vertice O(N) (worst case) with every vertice in bag_2 O(N) (worst case) performing a constant operation each time O(1) $\implies$ $O(N^3)$. Therefore we have (in worst case) run time of $O(N + N^2 \cdot D + N^3)$

Giving us a time complexity of $\mathbf{O(N^2 \cdot (N + D))}$

□

6.5 An Example

The example for this would be terrible. I can come in and discuss it fully if my explanaition was lacking at all. Please do not hesitate to ask.