John Wang April 13, 2012

6.046 Takehome Quiz

R08 - Friday 2 PM - Instructor: Sarah

Problem	Title	Points	Grade	Initials
1	Bracelet Division	25		
2	Dividing the Delta Quadrant	25		
3	A First World Problem	25		
4	Social Networking	25		
Total		100		

1 Bracelet Division

A band of k thieves has stolen a bracelet adorned with t different types of jewels. They want to divide the bracelet fairly between them by cutting the bracelet in a small number of places and dividing the parts. For each type $i = 1, 2, \ldots t$, if there are c_i jewels of this type, then each thief wants at least $\lfloor c_i/k \rfloor$ of the jewels.

The thieves ask Professor X for help. The professor models the problem as a continuous problem: the open bracelet is the unit interval, and it is divided into sub-intervals that represent consecutive jewels of each type. The professor finds a way to partition the "continuous bracelet" fairly between the thieves using $(k-1)t$ cuts. However, those cuts do not necessarily correspond to legitimate cuts of the real bracelet, as they may cut across jewels, crushing them.

Given the cuts suggested by the professor, show how to efficiently compute $(k-1)t$ legitimate cuts that divide the actual jewel bracelet between the thieves to their satisfaction.

Executive Summary

This algorithm turns the jewel cutting problem into a decimal rounding problem. The decimal parts provided by Professor X's cuts can be rounded to the nearest integer to create a satisfactory and legitimate group of cuts. This is done with max-flow and Ford-Fulkerson. The algorithm runs in $O((k+t)^5)$ time.

Algorithm

Take the cuts given to us by Professor X and create the matrix A composed of a_{ij} which gives the number of jewels of type i taken by thief j. Thus, there will be t rows, one for each type of jewel, and k columns, one for each thief, in the matrix A. Now, create the matrix A' which is composed of $a'_{ij} = a_{ij} - \lfloor a_{ij} \rfloor$ and gives the decimal part of each element in A.

Next, create an auxiliary matrix D of the same size as A, initially setting all $d_{ij} = 0$. Also, set γ_j as the sum of the jth column in A'. Search through the matrix A and if there exist any a_{ij} such that $a_{ij} < \lfloor c_i/k \rfloor + 1$, then set $d_{ij} = 1$ and reset $\gamma_j = \gamma_j - (1 + \lfloor c_i/k \rfloor - a_{ij})$. Do this for all $i \in \{1, 2, \ldots, t\}$ and $j \in \{1, 2, \ldots, k\}$.

Now we create a directed graph G with a starting and ending nodes s and e respectively. There will also be nodes x_i for all $i \in \{1, 2, \ldots, t\}$ and nodes y_j for all $j \in \{1, 2, \ldots k\}$. Here, x_i will correspond to the different types of jewels in the rows of A' and y_j will correspond to the different thieves in the columns of A'. The capacities will be set at

- $c(s, y_j) = \gamma_j$ for all $j \in \{1, 2, \ldots, k\}$
- $c(y_j, x_i) = 1$ for all $i \in \{1, 2, \ldots, t\}$ and $j \in \{1, 2, \ldots, k\}$ for which $d_{ij} = 0$
- $c(x_i, e) = c_i$ where c_i is the total number of bracelets of type i for all $i \in \{1, 2, \ldots, t\}$

All other edges will be set to 0. Now we find a max flow on the graph G from s to e. We use flows from the max flow and augment matrix D. If $d_{ij} \neq 1$, then we take the value of the flow from nodes x_i to y_j and add them to d_{ij}. This updating corresponds to $d_{ij} = d_{ij} + f(x_i, y_j)$ if $d_{ij} \neq 1$. Now, we create a new matrix $A^* = A' + D$ which shall replace the previous A. This new A^* shall have integer elements. To obtain the new cuts, examine the difference between the matrices $A^* - A$. This matrix shows which cuts to expand, and which to contract. If the ijth element of $A^* - A$ is negative, then shrink the size of the Professor's cut corresponding to the ijth element of A, and vice versa for positive. Doing this for all i and j provides the new cuts.

Correctness

First we shall show that there is a satisfactory group of legitimate cuts within 1 jewel of Professor X's proposed cuts.

Theorem 1 *If there exists a satisfactory group of legitimate cuts, then there exists a satisfactory group of legitimate cuts where each cut in the group is within at least one jewel of Professor X's proposed cuts.*

Proof. Suppose not, by contradiction. Let a_{ij} be the number of jewels of type i that the jth thief has. Then there must exist some a_{ij} where $a_{ij} < \lfloor c_i/k \rfloor$ for any combination of cuts within one jewel of Professor X's proposed cuts. First, let us show the contradiction when there is only one a_{ij} such that $a_{ij} < \lfloor c_i/k \rfloor$. Notice that if $k|c_i$, then we are done because the professor can only make cuts in between jewels (or else each thief would be unable to obtain exactly c_i/k jewels). This means that we only need to prove the case when there are $c_i - k\lfloor c_i/k \rfloor$ leftover jewels to be distributed.

Since the number of jewels that person i has must be an integer, we must have $a_{ij} \leq \lfloor c_i/k \rfloor - 1$. Moreover, since this jewel must go somewhere, there must be some person j' for which $a_{ij'} > \lfloor c_i/k \rfloor + 1$. If this is the case, then person j' has an extra jewel of type j' in one of his bands. Moreover, for one of these bands, one extra jewel of type i must be located at the end of the band next to the cut. This is because Professor X's cut is satisfactory for the thieves, so that one can always shift the cuts over by at most one jewel in order to obtain a jewel of type i. Once this is accomplished, since we know that all other a_{ij} have the property that $a_{ij} > \lfloor c_i/k \rfloor$, we can move the cut over by 1 jewel so that the extra jewel of type j' is taken up by another cut satisfying $a_{ij} > \lfloor c_i/k \rfloor$ for all its jewels. Then, we can give this

band to thief j and give his previous bands to the other thief. This will allow everyone to be satisfied. Thus, we have proven the theorem for one jewel.

Now suppose multiple jewels have the property $a_{ij} < \lfloor c_i/j \rfloor$. We can repeatedly perform the same process in order to satisfy all the thieves for jewel i. After the thieves are satisfied with the number of jewels they have for type i, we perform the process on the next type. Continual application of this allows all the thieves to be satisfied for an arbitrary number of jewels where $a_{ij} < \lfloor c_i/j \rfloor$. Thus, there exists a satisfactory and legitimate set of cuts where we move at most one jewel to the right or left of Professor X's proposed cut, for each of the Professor's cuts. This proves the theorem. □

Now, we know that if there exists any $a_{ij} < \lfloor c_i/k \rfloor + 1$, then taking the lower cut (the cut such that a_{ij} is less than in Professor X's solution) will result in $a_{ij} < \lfloor c_i/k \rfloor$. Therefore, we must choose the higher cut in this case. Thus, in the matrix D which tells us whether to round up or down, we have artificially set $d_{ij} = 1$ because we know we must round upwards, or else we will not be able to satisfy the criterion for a cut. Thus, we also set the new a_{ij} automatically to round up, and we can remove it from our graph G. We do this by setting $f(x_i, y_j) = 0$ and removing the difference between $\lfloor c_i/k \rfloor + 1$ and a_{ij} from the column sum, which results in the expression $\gamma_j = \gamma_j - (1 + \lfloor c_i/k \rfloor - a_{ij})$.

Theorem 2 *If f is the max flow on the graph G, then $f(x_i, y_i)$ will equal either 0 or 1 and will give the correct rounding for a group of satisfactory and legitimate cuts.*

Proof. The first part of the theorem, that $f(x_i, y_i) = 0, 1$, can be shown by the integrality theorem, which states that the maximum flow must be an integer when all capicities are integers (which is the case here). Moreover, since the capacity of $c(x_i, y_i) = 1$, we know that the flow can only take on values 0 or 1.

Next, we show that the the flow $f(x_i, y_i)$ gives the correct rounding so that $A^*_{ij} = a_{ij} - \lfloor a_{ij} \rfloor + f(x_i, y_i)$ provides the basis of a legitimate and satisfactory group of cuts. First, we know that a legitimate group of cuts is one for which all A^*_{ij} are integers. Clearly, since $f(x_i, y_i)$ takes on integer values, this condition will hold. Next, we know that a satisfactory group of cuts comes about when all $A^*_{ij} > \lfloor c_i/k \rfloor$ for all i and j. However, we know that if $a_{ij} < \lfloor c_i/k \rfloor + 1$, then we have already set $d_{ij} = 1$ and have disregarded it in G. This means that for any flow $f(x_i, y_j)$, one will always find that $A^*_{ij} = a_{ij} - \lfloor a_{ij} \rfloor + f(x_i, y_i) > \lfloor c_i/k \rfloor$. Thus, if we round down and set $f(x_i, y_j) = 0$, we are never in danger of letting A^*_{ij} become lower than $\lfloor c_i/k \rfloor$. Thus, the max flow from $f(x_i, y_i)$ will give the correct rounding for a group of satisfactory and legitimate cuts. □

Running Time Analysis

The algorithm requires $O((k-1)t) = O(kt)$ time to contsruct A from the Professor's recommendations. Next, creating A' will take $O(kt)$ to loop over all the elements a_{ij} and to

subtract of $\lfloor a_{ij} \rfloor$ for all $i \in \{1, 2, \ldots, t\}$ and $j \in \{1, 2, \ldots, k\}$. Creating the matrix D will also require $O(kt)$ time because we are again looping over a_{ij}. The updating of γ_j should take $O(1)$ each time for at most $O(kt)$ updates, so in total it will take $O(kt)$. Finally creating G requires $O(k + t)$ nodes, which takes $O(k + t)$ time. Running a max-flow algorithm such as Ford-Fulkerson requires $O(VE^2) = O((k + t)^5)$ time.

Reconstructing the cuts takes $O(kt)$ time by running through all the cuts given by Professor X. Thus, the entire algorithm requires $O(kt+kt+kt+kt+(k+t)+(k+t)^5+kt) = O((k+t)^5)$ time.

2 Dividing the Delta Quadrant

In year 3512, the Star Federation has finally reached a consensus in the century long conflict between Treaps and Cycles for the asteroid field in the Delta Quadrant. To avoid any tension between the two factions, the decision has been made to divide the asteroids in a way that maximizes the minimum distance between any two asteroids assigned to different factions, even if the number of asteroids given to each is uneven (but no faction shall be left without any asteroids).

Design an efficient algorithm which, given a list of n asteroid coordinates (a 3-tuple of real numbers) in the Delta Quadrant, returns the optimal partition maximizing the minimum distance between any two asteroids assigned to different partitions.

Executive Summary

A dynamic programming solution is presented which requires $O(n^2)$ running time. We use an arbitrary ordering of the asteriods which allows us to find the maximum minimum distance between any two asteriods for a subset $\{1, 2, \ldots, i\}$ of the n asteriods. Then, we use this previous result to obtain our next maximum minimum distance for $\{1, 2, \ldots, i+1\}$ by trying to place asteriod $i + 1$ in both groups 1 and 2, and taking the maximum of the minimum distances of each case. Note that this algorithm can be easily extended to work for k different groups, and will have a running time of $O(kn^2)$.

Algorithm

We shall use a dynamic programming solution. First, order the asteriods arbitrarily so that each asteriod is assigned some $i \in \{1, 2, \ldots, n\}$. Define $P(i)$ as a dictionary of the positions in (x, y, z) coordinates of the asteriods in the set $\{1, 2, \ldots, i\}$. Next, we define $DP(i, g)$ as the maximum minimum distance for the asteriods in the set $\{1, 2, \ldots, i\}$, given that the ith asteriod is in group g. Since there are two groups, g can take on values 1 or 2. Also, we shall cache the groupings of the asteriods corresponding to $DP(i, g)$ using $L(i, g)$, which is a list of the first i asteriods and their group (EX: L(i, g) = [1, 2, 2, 1, 1, 1, ...]). Now we shall have the following recursive relation:

$$DP(i, g) = \max \begin{cases} \min\{DP(i-1, 1), minDist(i, g, P(i), L(i-1, 1))\} \\ \min\{DP(i-1, 2), minDist(i, g, P(i), L(i-1, 2))\} \end{cases} \tag{1}$$

Where $DP(2, 1) = DP(2, 2)$ is the distance between asteriods 1 and 2 and $minDist(i, g, L)$ computes the minimum distance between i in group g any point x in group $g + 1 \pmod 2$ using the positions given in $P(i)$. The algorithm should start with $i = 2$ and increment until $i = n$, computing using $g = 1, 2$ at each i. The final result will be $\max\{DP(n, 1), DP(n, 2)\}$.

The $minDist(i, g, P, L)$ algorithm takes the minimum of $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$ for all j where $L(j) \equiv g + 1 \pmod 2$.

Correctness

Next, we shall show that the dynamic programming algorithm is correct. We must prove that $DP(i, g)$ really is the maximum minimum distance for the asteriods in the set $\{1, 2, \ldots, i\}$ given that the ith asteriod is in group g. For $i = 2$, we see that $DP(2, 1)$ and $DP(2, 2)$ must be correct because they simply involve two asteriods, whose maximum minimum distance is clearly just their distance.

Now suppose $DP(k, g)$ is correct for $k < i$, and consider $DP(i, g)$ as set forth by the recursion above. We see that when we add the ith asteriod into the comparison, we need to add it to a previous set of $i - 1$ asteriods that have already been grouped. To find the new maximum minimum distance, we must add i to the previous maximum minimum distance set, otherwise, we could just increase the minimum distance to that maximum. Moreover, we know that when we add the ith asteriod into the set, there are two options. First, the minimum distance between i and an asteriod in a different group could be greater than $DP(i - 1, 1)$. In this case, the minimum distance stays the same. The second case is when the minimum distance between i and an asteriod in a different group is less than $DP(i-1, 1)$. In this case, we want to set the new minimum distance to $minDist(i, g, P(i), L(i - 1, 1))$. This is why we take $\min\{DP(i - 1, 1), minDist(i, g, P(i), L(i - 1, 1))\}$. Moreover, since i can be placed in groups 1 or 2, we want to take the maximum over the minimums.

Since we are taking a maximum over the best possibilities, this algorithm will give the maximum minimum value. We know these are the best possibilities because $DP(i - 1, g)$ is as large as possible. In the case when $minDist(i, g, P(i), L(i - 1, 1)) < DP(i - 1, 1)$, the distance $DP(i - 1, 1)$ doesn't matter. But when the opposite is the case, we want to make $DP(i - 1, 1)$ the largest we possibly can make it, which occurs exactly when we select the maximum minimum distance which we chose previously of $DP(i - 1, g)$. This shows that $\max\{DP(n, 1), DP(n, 2)\}$ will give the maximum minimum distance for any partition of asteriods into two groups.

Running Time Analysis

First, we know that we must look of $i \in \{2, 3, \ldots, n\}$, and for each i, we must loop over $g = \{1, 2\}$ in order to obtain $DP(n, 1)$ and $DP(n, 2)$. This means we must perform the maximum operation $O(2n) = O(n)$ times. It takes $O(n)$ time for $minDist(i, g, P(i), L(i - 1, 1))$ because it computes $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$, which requires $O(1)$ time, over at most $O(n)$ items. Taking the minimum over n items requires a loop over n items. The total cost of $minDist$ is therefore still $O(n)$. Taking the minimum of $DP(i - 1, 1)$ and $minDist(i, g, P(i), L(i - 1, 1))$ therefore takes $O(n)$ time because $DP(i - 1, 1)$ has $O(1)$

access time, and computing maxima between two elements takes $O(1)$ time as well.

Each step in the loop over i and g therefore takes $O(n)$ time. Thus, the entire recursion requires $O(n \cdot n) = O(n^2)$ time. The maximum at the end of the algorithm requires $O(1)$ time and initialization of the loop also requires $O(1)$ time to calculate the distance between asteriods 1 and 2. This means the entire algorithm runs in $O(n^2 + 1) = O(n^2)$.

Also note that the algorithm requires $O(n)$ space. Since on the ith iteration of the algorithm, we only need values from the $i - 1$st iteration of the algorithm and the dictionary P. The dictionary P takes up $O(n)$ space, $DP(i-1, 1)$ and $DP(i-1, 2)$ require $O(1)$ space, and $L(i-1, 1)$ and $L(i-1, 2)$ both require $O(n)$ space. In total, this adds up to $O(n)$ space for the entire algorithm.

3 A First World Problem

Billy Billionaire wants to create an artificial lake in the mountains near his mansion. Fortunately, he lives in 2D land, which makes the problem much easier. He has elevation data, taken at regular intervals across the two-dimensional mountains. We can write this elevation data as $A[1], \ldots, A[n]$. *The stretch of land from* i *to* j *can be made into a lake if and only if for all* $i < k < j$, $A[k] < A[i]$ *and* $A[k] < A[j]$. *We say that the height of the lake* $[i, j]$ *is equal to* $\min\{A[i], A[j]\}$, *and the width of the lake* $[i, j]$ *is equal to* $(j - i - 1)$.

Billy wants to pick the best stretch of land $[i, j]$ *on which to build a lake. He wants the lake to be very wide, so that it looks impressive. However, the lake height should be relatively low, so that his fleet of helicopters have an easier time airlifting the water to the new lake. After some contemplation, Billy decides that he wants to pick a stretch of land* $[i, j]$ *that can be made into a lake that maximizes* $(\text{lakewidth} - \frac{1}{100} \cdot \text{lakeheight})$. *Design an efficient algorithm that finds a lake maximizing Billy's chosen objective function.*

Executive Summary

A greedy minimum finding algorithm which requires $O(n \lg n)$ time is presented. The algorithm makes use of a binary search tree structure to recursively find the minimum and create a new, artificially modified minimum until the search tree is empty. The algorithm finds all the possible lakes, then, from the resulting list of lakes, finds the lake the maximizes the objective funciton.

Algorithm

The algorithm first finds all the lakes, then computes the best lake according to the objective function. To initiate the algorithm, all $A[i]$ are inserted into a binary search tree as keys, and fields for the left and right endpoints of the $A[i]$ are kept in the node object. The left endpoint is initialized as $i - 1$ and the right endpoint as $i + 1$. Next, the algorithm follows four general steps to find all the lakes.

First, it finds the current minimum value and removes it from the tree. Second, if the current minimum is not an endpoint of the array, the algorithm stores the left and right endpoints of the minimum value as a lake. Third, a new tree node is created with key value set as the minimum of the left and right endpoints of the minimum value. The left endpoint of this new node is the smallest value l such that for all $l \leq k \leq m_l$, $A[k] = A[m_l]$ were m_l is the left endpoint of the minimum node. The right endpoint of the new node is similarly defined as the greatest value r such that for all $m_r \leq k \leq r$, $A[k] = A[m_r]$ where m_r is the right endpoint of the minimum node. This process will also delete all values $A[k]$ from the binary search tree where $l < k < r$. Finally, we recurse on the first step until the binary search tree

no longer has any nodes.

The algorithm then looks for the best lake where the objective function is defined as $r - l - 1 - \frac{1}{100}\min\{A[l], A[r]\}$ for a lake (l, r). It does this by looping through all the lakes found in the previous step, computing the objective function, and keeping a running count of the maximum. Python pseudocode for the procedure is given below:

```
def LakeAlgorithm():
    BST = BinarySearchTree()
    for i in range(n):
        NewNode = BSTNode()
        NewNode.key = A[i]
        NewNode.left = i - 1
        NewNode.right = i + 1
        BST.insert(NewNode)

    lakes = []
    while BST.empty() == False:
        m = BST.PopMin()
        l = m.left
        r = m.right
        if not (BST.isMinIndex(l) || BST.isMaxIndex(r)):
            nextmin = min(A[l], A[r])
            while A[l] == nextmin:
                BST.remove(A[l])
                l = l - 1
            while A[r] == nextmin:
                BST.remove(A[r])
                r = r + 1
            NewNode = BSTNode()
            NewNode.key = nextmin
            NewNode.left = l
            NewNode.right = r
            BST.insert(NewNode)
            lakes.append((l,r))

    max_objective = 0
    best_lake = None
    for (l, r) in lakes:
        if ObjectiveFunction(l, r) > max_objective:
            max_objective = ObjectiveFunction(l, r)
            best_lake = (l, r)

    return (l,r)
```

```
def ObjectiveFunction(l, r):
    return r-l-1-(1/100)*min(A[l], A[r])
```

Note that in order to implement $BST.isMinIndex(l)$ and $BST.isMaxIndex(r)$, the BinarySearchTree object keeps a supplementary Binary Search Tree available which keeps track of all Node.left and Node.right values. This makes it easy to check if l or r is a minimum or maximum index, respectively.

Correctness

First, we shall note that if $A[i]$ is a unique minimum in the set $A[1], \ldots, A[n]$ and it is not an endpoint, then there exists a lake from $A[i-1]$ to $A[i+1]$. This is because $A[i] < A[k]$ for all $k \in \{1, \ldots, i-1, i+1, \ldots n\}$. Therefore, it surely must be the case that $A[i] < A[i-1]$ and $A[i] < A[i+1]$, which constitutes the definition of a lake. Now we shall prove that the modified minimums that the algorithm work with also produce lakes. Define a modified minimum as a minimum that was inserted into the BST in the LakeAlgorithm.

Theorem 3 *If $M[l, \ldots, r]$ is a modified minimum over the subarray $A[l], \ldots, A[r]$, $A[l] \neq A[1]$, and $A[r] \neq A[n]$, then there exists a lake from $A[l]$ to $A[r]$.*

Proof. A modified minimum $M[l, \ldots, r]$ has the property that $\max\{A[l], \ldots, A[r]\} < A[i]$ for all $i \notin \{l, \ldots, r\}$. Since the modified minimum is not on an endpoint because $A[r] \neq A[n]$ and $A[l] \neq A[1]$, we know that $A[l]$ and $A[r]$ both exist and have the property $A[l] > \max\{A[l], \ldots, A[r]\}$ and $A[r] > \max\{A[l], \ldots, A[r]\}$. This is exactly the definition of a lake, which completes the proof. □

Theorem 4 *If $A[L]$ to $A[R]$ constitutes a lake, then there exists a modified minimum $M[l, \ldots, r]$ where $L < l$ and $r < R$.*

Proof. If $A[L]$ to $A[R]$ is a lake, then we must have $A[k] < \min\{A[L], A[R]\}$ where $L < k < R$. This means that we can take $\max_{L<k<R}\{A[k]\}$ to be the value of the modified, and take $l = L + 1$ and $r = R - 1$, to create a modified minimum $M[l, \ldots, r]$. □

Theorem 5 *The LakeAlgorithm finds all possible modified minima.*

Proof. Suppose $M[l, \ldots, r]$ is a modified minimum such that $\max\{A[l], \ldots, A[r]\} < A[i]$ for all $i \notin \{l, \ldots, r\}$. Then there are two cases. In the first case, all k for which $l < k < r$ have the property that $A[l] = A[k] = A[r]$, in which case the LakeAlgorithm will eventually

select this as a minimum as it goes along and recombines minima. In the second case, theres exist some k for which $A[k] < A[l]$ and $A[k] < A[r]$ where $l < k < r$. In this case, the LakeAlgorithm will take that $A[k]$ as a minimum before it finds the $M[l, \ldots, r]$ and recombine it. In a chain of recursive calls of recombining, we will eventually end up with $A[l] = A[k] = A[r]$ for all $l < k < r$. This goes back to the first case. Thus, the LakeAlgorithm will find a modified minimum if one exists. □

Thus, the LakeAlgorithm will only add actual lakes to the list of lakes by the first theorem. Next, by the second and third theorems, the LakeAlgorithm will find all possible modified minima, and therefore, all possible lakes. Since the algorithm finds all possible lakes and finds only lakes, we know the algorithm must be correct when it finds the maximum objective function value over all the lakes in the lake list. Therefore, the algorithm must correctly find the best possible lake for Billy.

Running Time Analysis

To analyze the runtime, we refer back to the python code. The initialization at the beginning requires $O(n \lg n)$ time to insert n new nodes into the BST. Next, the algorithm continues until the BST is empty. During each iteration of the loop, the algorithm performs a PopMin operation requiring $O(h)$ time, checks to see if l and r are minimum or maximum indices in $O(1)$ time, walks down the list of remaining values in A in $O(BST.Size())$, and inserts a new BST node in $O(h)$ time.

However, notice that at any given time $h = O(\lg n)$ because there can be at most n nodes in the tree at any given time. Moreover, we note that if the algorithm walks through k_l values of $A[l]$ to the left and k_r values of $A[r]$ on the right, the new BST will have $n - k_l - k_r$ nodes after the recombine. Therefore, the loop until the BST is empty takes at most $O(n)$ time. Moreover, if the walk down the list takes $O(BST.size())$ time, the loop will finish. Therefore, we see that the walking down the list takes $O(n)$ time in total over the entire loop, which means it takes $O(1)$ amortized time on average during the loop. All of this together, means that the loop takes $O(n \lg n)$ time.

Finally, finding the maximum objective function over all the lakes takes time $O(l)$, where l is the number of lakes. This is because it takes $O(1)$ time to evaluate the objective function for each lake. However, we know that there are at most $O(n)$ lakes because the loop in the LakeAlgorithm takes $O(n)$ time to finish, and it finds a maximum of one lake during each pass. Since it finds all possible lakes (as shown from the theorems from the previous section), we know there are $O(n)$ lakes.

The total runtime of the function is therefore $O(n \lg n) + O(n \lg n) + O(n) = O(n \lg n)$.

4 Social Networking

MITbook+ is a new social networking site created by MIT students. Much like other social networking sites, MITbook+ allows you to establish "friendship" with other users. Each user has a unique MITbook+ ID number, and a list of all MITbook+ ID numbers is available to the general public. Furthermore, for any pair of user IDs, there is a publically-accessible webpage that can be used to check whether the pair are friends. However, in the interests of preserving user privacy, all other information associated with MITbook+ IDs is available to members only.

Currently, you are not a user of MITbook+, but you want to learn more about a specific person (known to you by his ID as 60461337). Specifically, you want to determine whether he went to school at MIT. Fortunately, the founders of MITbook+ have posted some statistics on their blog that may help you. Currently, MITbook+ has N users (a very large number). Exactly $2N/3$ users attended MIT; the other $N/3$ did not. Interestingly, because MIT is a very tight-knit community, every current or former MIT student on MITbook+ is friends with every other current or former MIT student on MITbook+.

Part (a)

In the blog post with the statistics, the MITbook+ founders originally claimed that every user on MITbook+ who did not attend MIT was friends with at most $N/4$ users who did attend MIT. Create an efficient algorithm that uses this fact to determine whether user 60461337 attended MIT.

Executive Summary

This $O(N)$ deterministic algorithm counts the number of friends that 60461337 has. Since someone goes to MIT if and only if they have more than $2N/3$ friends, a simple algorithm is developed to see whether the number of friends is greater than $2N/3$.

Algorithm

For each ID i in the list of ID's except for 60461337, check if $Friends(i, 60461337) == True$. Keep a running count of the number of trues. At the end of the algorithm, if $Count \geq 2N/3$, then the algorithm returns that 60461337 attended MIT. Otherwise, return that he did not attend MIT.

Correctness

This algorithm will always be correct because we can be sure that only MIT students can have a friend count of greater than $2N/3$. First, it is clear that all MIT students have at least $2N/3$ friends, because all MIT students are friends with each other and there are $2N/3$ MIT students.

Next, we see that if someone is a non MIT student, then he can be friends with at most $N/4$ MIT students. Since there are $N/3$ non MIT students, this non MIT student can be friends with at most $N/3 + N/4 = 7N/12$ people. Clearly $7N/12 < 2N/3$. Thus, someone goes to MIT if and only if he has more than $2N/3$ friends and someone does not go to MIT if and only if they have less than $7N/12$ friends. This shows that the algorthim will always return the correct answer.

Running Time Analysis

The algorithm takes time $O(N)$ in order to determine the number of friends that 60461337 has. The algorithm must loop through the entire list of N people, checking whether i and 60461337 are friends in $O(1)$ time. Therefore, the total running time is $O(N)$.

Part (b)

Not long after the original blog post, the founders updated their numbers. Apparently, the original claim spurred a large number of non-MIT users to friend more MIT students. Now, after this friending spree, every user on MITbook+ who did not attend MIT is friends with at most $2N/5$ *users who did attend MIT. This also had the effect of raising the average number of friends per user to exactly* $2N/3$*. Given this updated information, devise an efficient algorithm that checks whether user 60461337 went to school at MIT.*

Executive Summary

This deterministic algorithm runs in $O(n^2)$ time and always provides the correct answer. The algorithm is based on the fact that no more than $N/6$ non-MIT students can have more than $2N/3$ friends. Using this, the algorithm finds $N/6$ non-MIT students, then uses this new information to simplify the problem of determining whether 60461337 is an MIT student.

Algorithm

For each person i on MITBook+, count the number of friends they have. Do this by counting the number of times $Friends(i, j) == True$ for j ranging over all the IDs not equal to i in the list of IDs. Next, find the people who have fewer than $2N/3$ friends and add them to a set $NoFriends$.

Now, compute the number of friends of 60461337 who are not in the $NoFriends$ set. This can be done by looping over all $j \notin \{60461337\} \cup NoFriends$ and counting the number of occurences of $Friends(60461337, j) == True$. If the new number of friends is less than $2N/3$, then return that 60461337 is a non-MIT student. Otherwise if 60461337 has greater than or equal to $2N/3$ friends, then return that he is an MIT student.

Correctness

The correctness of this algorithm comes from the fact that only $1/2$ of the non-MIT students can have more than $2N/3$ friends. To see that this is true, we prove the theorem:

Theorem 6 *Only half of the $N/3$ non-MIT students can have greater than or equal to $2N/3$ friends.*

Proof. We want to find the largest proportion x of non-MIT students who have greater than or equal to $2N/3$ friends. Notice that the proportion x is maximized when all the non-MIT students who have greater than or equal to $2N/3$ friends have exactly $2N/3$ friends. If some non-MIT student had more than $2N/3$ friends, we could possibly increase x by giving those extra friends to someone else. Moreover, each non-MIT student should have the maximum number of friends possible between non-MIT students. Since there are a total of $N/3$ non-MIT students, each non-MIT student will have a maximal $N/3$ friends who are non-MIT students.

Since there will be $xN/3$ students with exactly $2N/3$ friends, $N/3$ of which are non-MIT friends, each of them will have $N/3$ MIT friends. Next, we know that all MIT students must be friends with all other MIT students, which means each has at least $2N/3$ friends. Moreover, we know that the total number of non-MIT students that MIT students are friends with must match the number of MIT students that non-MIT students are friends with. Since there are $x(N/3)$ non-MIT students each with $N/3$ MIT friends, the total number of non-MIT students that MIT students are friends with must be $x(N^2/9)$. The total number of friends is therefore:

$$\left(\sum_{i=1}^{N/3} \frac{N}{3} + \left(x\frac{N}{3}\right)\frac{N}{3}\right) + \left(\sum_{i=1}^{2N/3} \frac{2N}{3} + \left(x\frac{N}{3}\right)\frac{N}{3}\right) \tag{2}$$

The left parentheses represents the number of friends of non-MIT students, and the right side is the number of friends for MIT students. Since $2N/3$ is the average number of friends for any given person, we must have:

$$\frac{1}{N}\left[\left(\sum_{i=1}^{N/3}\frac{N}{3}+\left(x\frac{N}{3}\right)\frac{N}{3}\right)+\left(\sum_{i=1}^{2N/3}\frac{2N}{3}+\left(x\frac{N}{3}\right)\frac{N}{3}\right)\right]=\frac{2N}{3} \tag{3}$$

Simplifying this expression and noting that $\sum_{i=1}^{N/3} N/3 = N^2/9$ and $\sum_{i=1}^{2N/3} 2N/3 = 4N^2/9$, we can solve the following equation for x:

$$\frac{N^2}{9}+x\frac{N^2}{9}+\frac{4N^2}{9}+x\frac{N^2}{9} = \frac{2N^2}{3} \tag{4}$$

$$\frac{5N^2}{9}+x\frac{2N^2}{9} = \frac{2N^2}{3} \tag{5}$$

$$x = \frac{1}{2} \tag{6}$$

Thus, the maximum proportion x of non-MIT students who can have more than $2N/3$ friends is $1/2$. □

Now, we use the fact that if any person has less than $2N/3$ friends, they must be a non-MIT student, because all MIT students have at least $2N/3$ friends from MIT. Thus, all people who are in the *NoFriends* set must be non-MIT students. Moreover, we know there are at least $N/6$ people in this set. This is because at most $(1/2)(N/3)$ non-MIT students can have more than $2N/3$ friends. Thus, at least $N/6$ non-MIT students have less than $2N/3$ friends, and they constitute the *NoFriends* set.

Since we know that $N/6$ people are definitely non-MIT students, we have more information during our second count. If 60461337 is an MIT student, then subtracting away $N/6$ non-MIT students from his friend count (if they are his friends) will mean he still has at least $2N/3$ friends who are MIT students. Thus, if 60461337 is an MIT student, his new friend count will stay above $2N/3$.

However, if 60461337 is a non-MIT student, subtracting away $N/6$ non-MIT students from his friend count will imply that his new friend count is less than $2N/3$. This is because if 60461337 has the maximum $N/3$ non-MIT friends and the maximum $2N/5$ MIT friends, he will still only have $N/3 + 2N/5 - N/6 = 17N/30$ friends in his second count, which is less than $2N/3$. Note that 60461337 can have at most $N/3 - N/6$ non-MIT friends in his second friend count, because the $N/6$ people in *NoFriends* essentially shrinks the number of available non-MIT friends by $N/6$ people. Thus, even if 60461337 has the maximum number of MIT friends, he still cannot have more than $2N/3$ friends on the second count.

This shows that 60461337 will be an MIT student if and only if he has more than $2N/3$ friends on the second count. This shows that the algorithm will always produce the correct answer.

Running Time Analysis

The algorithm must find the number of friends for each person in the list of IDs. Finding the number of friends for any person i requires looping over all N IDs in the ID list and checking whether $Friends(i, j) == True$ in $O(1)$ time. Thus, finding the number of friends takes $O(N)$ time. Doing this for all N people requires $O(N^2)$ time.

Creating the $NoFriends$ set requires looping over all N people, checking their friend counts, and placing them into the set if necessary. If one uses a hash-table implementation of the set, this takes $O(1)$ time person person. Thus, creating the $NoFriends$ set requires a total of $O(N)$ time.

Next, finding the second friend count of 60461337 requires looping through all N IDs the ID List, checking if it belongs to the $NoFriends$ set in $O(1)$ for a hash-table, and counting the occurrence of $Friends(60461337, j) == True$ in $O(1)$ time. The total time for this is therefore $O(N)$.

Therefore, the entire algorithm requires $O(N^2 + N + N) = O(N^2)$ time.