

NATIONAL UNIVERSITY OF SINGAPORE

CS2040 – DATA STRUCTURES AND ALGORITHMS

(Semester 1: AY2019/20)

Time Allowed: 2 Hours

INSTRUCTIONS TO STUDENTS

1. Do **NOT** open the question paper until you are told to do so.
2. This assessment paper consists of **Twelve (12)** printed pages and **Nine (9)** questions with possible subsections.
Important tips: Pace yourself! Do **not** spend too much time on one (hard) question.
3. This is an **Open Book Assessment**. You can check the lecture notes, tutorial files, problem set files, CP3 book, or any other books that you think will be useful.
4. When this Assessment starts, **please immediately write your Student Number** below. Do not write your name.
5. You may write your answers in pencil except student number below which should be written with a pen.
6. All the best!

STUDENT NUMBER:

A								
---	--	--	--	--	--	--	--	--

(Write your Student Number above legibly with a pen.)

Questions	Possible	Marks
Q1-5	15	
Q6	25	
Q7	20	
Q8	15	
Q9	25	
Total	100	

Section A – Analysis (15 Marks)

Prove (the statement is correct) or disprove (the statement is wrong) the following statements below. If you want to prove it, provide the proof or at least a convincing argument. If you want to disprove it, provide at least one counter example. 3 marks per each statement below (1 mark for circling true or false, 2 marks for explanation):

1. After doing FindSet(x) operation in Union-Find Disjoint Sets data structure with the path compression heuristic activated, the real height (not the rank) of the disjoint set that contains x always decreases, when x is not the root of the disjoint set. **[True/False]**

False. If x is a child of the root then the real height will still not decrease since x is already placed under the root so path compression does not affect the height.

2. Given any AVL tree of height 4, deleting any vertex in the tree will not result in more than 1 rebalancing operation (not rotation but rebalancing operations!). **[True/False]**

False. Already have an example in the lecture notes which is an AVL of height 4 where deleting a vertex results in 2 rebalancing operations.

3. The time cost for changing a binary MAX heap to a binary MIN heap is $O(n)$. **[True/False]**

True. Just treat the array containing the max heap as input and call $O(N)$ fast heap create on it to get the min heap.

4. To form the MST of an undirected connected graph, exactly 1 edge is removed from every cycle in the graph, namely the largest edge in the cycle. **[True/False]**

False. In fig 1 below, all the edges in the cycle 0,1,3,4,0 will be removed to get the MST in fig 2.

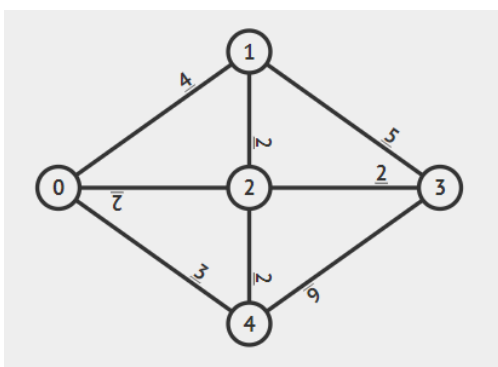


Fig 1

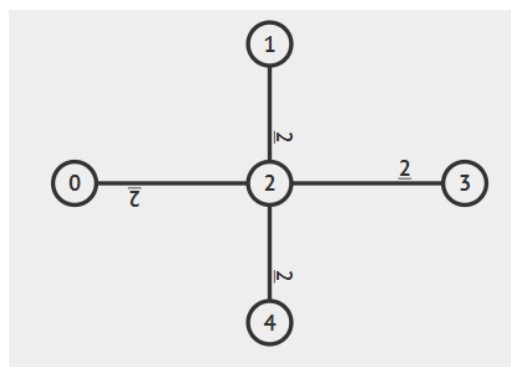


Fig 2

5. It is possible for every vertex in a DAG to have an outgoing edge to some other vertex in the DAG. [True/False]

False. Argument is similar to the proof that there must be a vertex with no incoming edges in a DAG as given in the lecture notes.

If every vertex in a DAG has an outgoing edge that means, we can pick any vertex X and choose an outgoing edge from X and move to the next vertex from that vertex we can do the same since every vertex has at least one outgoing edge. After we have visited more than N vertices like this where N is the number of vertices in the DAG, then 1 vertex Y must have been repeated. The edges used between the 2 successive visits of Y will form a cycle, and this is a contradiction that the graph is a DAG.

For section B, Partial marks will be awarded for correct answers which do not meet the time complexity required (or if the required time complexity of not given, then any correct answer that is less efficient than the lecturer's answer) . You can write in pseudo-code. Provide enough details to all user defined DSes/algorithms/modifications to taught DSes and algorithms so as to show understanding of the solution. *Sub-questions marked with * can potentially be more difficult.*

Section B – Application (85 Marks)

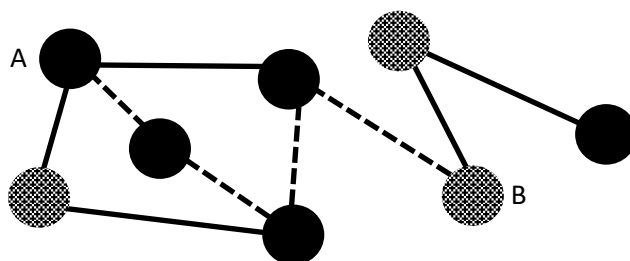
6. Battle Nations [25 marks]

The world of Aquarius is at war! This world is made up of mostly water with N islands which are connected by M bridges where $(N - 1 \leq M \leq \frac{N*(N-1)}{2})$. You may assume that there is always a way to get from any island to any other island. Each island belongs to either country **X** or country **Y**. These two countries are now at war with each other.


In order to attack an island of the enemy country, the attacking country must choose an island A that it owns to initiate the attack and an island B of the enemy country to attack.


Any **valid path** from A to B the attacker can choose must start from A and end at B such that B is the **first enemy island** encountered along the path (otherwise B cannot be attacked first without attacking the previous enemy island(s) along the path!).

Any bridge along a valid path may potentially be targeted by the enemy country's aerial bombers to be destroyed so as to prevent the attack from happening. Each bridge is given an integer value W ($1 \leq W \leq 10,000$) indicating its difficulty to be destroyed (**larger** W means it is more difficult to be destroyed).



Legend:

 = Island belonging to country X

 = Island belonging to country Y

The figure above is a small example of the world Aquarius, where the circles are the islands and the lines are the bridges connecting the islands. Here country X is the attacker selecting island A to attack country Y's island B. The dotted edges show a possible valid path to get from A to B.

**read b) and c) first before answering a)!*

a) If you want to represent the world of Aquarius as a graph $G(V, E)$ to answer part b) and c),

i.) What do the vertices in V represent and what information will you store in a vertex? **[2 marks]**

vertices are the islands. Use integer values 0 to N-1 to label each island. Have an attribute nation to indicate which nation the vertex belongs to. E.g 1 for country X and 2 for country Y

ii.) What do the edges in E represent and what information will you store in an edge? **[2 marks]**

edges are the bridges that link islands. For each edge we will store the weight which is the number indicating its difficulty in being destroyed.

iii.) Is this a directed or undirected graph? **[1 mark]**

This is an undirected graph since the bridges can be crossed both ways.

iv.) What is the best graph data structure D to store this graph to answer b)? **[1 mark]**

Adjacency list or edge list (depending on what is the MST algorithm used in b)).

- b) Given the graph data structure D as modelled in a), island A of an attacker and island B of the enemy to attack, give the best algorithm you can think of to find the best valid path to get from A to B such that the weakest bridge (based on its difficulty value of being destroyed) is **maximized** and output the difficulty value of that weakest bridge.
If there is no valid path from A to B output -1. [9 marks]

This is basically a maximin problem with a slight modification.

- 1.) Have a variable **weakest** that keep tracks of the weight of the weakest bridge in the best valid path from A to B . Initialize weakest to -1.
- 2.) Run Kruskal's algorithm on D but instead of sorting in ascending order, sort by descending order (to get a maximum spanning tree)
- 3.) Next, modify Kruskal's algorithm as follows:

```

For each edge (u,v,w) in sorted edge list
  If (u.nation == A.nation && v.nation == A.nation) // edge connects attacker islands
    If !UFDS.isSameSet(u,v)
      UFDS.unionSet(u,v)
      weakest = w
  else if ((u.nation == A.nation && v == B) ||
           (v.nation == A.nation && u == B)) // edge connects attacker island to B
    If !UFDS.isSameSet(u,v)
      UFDS.unionSet(u,v)
      weakest = w
  If UFDS.isSameSet(A,B) // found best valid path from A to B
    Break out of for loop

```

- 4.) output **weakest** if UFDS.isSameSet(A,B) else output -1

Same time complexity as standard Kruskal's : $O(M \log N)$

- c) Assuming C is the set of islands owned by the attacker and given a fixed enemy island B . For each of the islands A' in C with a valid path to B , the best path (path that maximizes the minimum edge) has been found and edges in the best path from A' to B has been put into an edge list EL . However, the enemy has just destroyed a weakest bridge (u', v') along the best path from an attacker island A to B , so the edge representing that bridge has been removed from EL . Given D and EL , now provide an algorithm to find the next best valid path from A to B in $O(N + M)$ time. Specify any modifications you need to make to the graph $G(V, E)$ and D and any extra data structures you use.
Again, if there is no more valid path output -1, otherwise output the weight of the weakest bridge in that next best valid path. [10 marks]

Here EL will store a subtree of the spanning tree that consists of only the edges from each attacker island to B.

Since the weakest bridge from an attacker island A to B is destroyed this will cause the subtree to be broken into 2 components.

Modify the vertices to contain an integer attribute **label** which will indicate which component the vertex belongs to.

Since EL is an edge list, create an adjacency list D' so that counting component algorithm can be run efficiently.

1.) Creating adjacency list from edgelist $\rightarrow O(N+M)$

a.) Create an empty adjacency list D' $\rightarrow O(N)$

b.) For each edge (u,v,w) in EL $\rightarrow O(M)$

Add (u,w) to neighbor list of v $\rightarrow O(1)$

Add (v,w) to neighbor list of u $\rightarrow O(1)$

(Optional can simply use Adj List in step 6)

Similarly, if D is an adjacency list need to convert it to an edge list as follows:

2.) Creating edge list from adjacency list $\rightarrow O(N+M)$

a.) Create an empty edge list EL' $\rightarrow O(1)$

b.) For each index v in D

For each neighbor (u,w) in neighbor list of v

Add (u,v,w) into EL'

3.) Run counting component algorithm on D' to label the 2 components. $\rightarrow O(M+N)$

4) (Corner case if you get a spanning graph in EL with cycle involving A and B and not a spanning tree)

if step 3 gives only 1 component that means there is a cycle and there is another path from A to

B with the same weakest bridge so simply return weight of deleted edge

5.) let bestedge = -1

6.) For each edge (u,v,w) in EL' or D (if already edge list) $\rightarrow O(M)$

If (u.label != v.label && (u,v) != (u',v')) $\rightarrow O(1)$ // possible replacement

If w > bestedge

bestedge = w

7.) return bestedge

Total time taken = $O(N+M)$

7. Sorting or Not? [20 marks]

- a) Given an **unsorted** array A of N unique **floating point** values of no fixed precision, where each value may only differ from its correct position in the **sorted** array by no more than K positions (K is much smaller than N), give an algorithm that will fully sort A in $O(N \log K)$ time. For example if $K = 3$ and in sorting A , the value 10.3 is at index 10, then in the original unsorted array 10.3 can be possibly found at indices 7,8,9,10,11,12,13. [10 marks]

Use another array A' and a min heap H as follows:

- 1.) insert the 1st $K+1$ values of A into $H \rightarrow O(K \log K)$
- 2.) for $i = K$ to $N-1 \rightarrow O(N)$
 - add $H.\text{extractMin}$ to back of $A' \rightarrow O(\log K)$
 - $H.\text{insert}(A[i]) \rightarrow O(\log K)$
- 3.) while ($!H.\text{empty}()$) $\rightarrow O(K)$
 - add $H.\text{extractMin}$ to back of $A' \rightarrow O(\log K)$

total time complexity = $O(K \log K) + O(N) * (O(\log K) + O(K \log K)) = O(N \log K)$ since $K < N$

- b) Given a K by N matrix L , where each row represents a sorted array of N unique **floating point** values of no fixed precision (thus K sorted arrays), give an algorithm to merge them into 1 sorted array of $K * N$ elements in $O(K * N * \log K)$ time. [10 marks]

- 1.) Let A be the final sorted array
- 2.) Let A' be an array of size K containing current indices of the K arrays during the sort. Initialize A' to 0.
- 3.) Let H be a min heap containing pairs (i, v) where i is the i th array and v is the current value under consideration in i . H is keyed using v
- 4.) For $i = 0$ to $K-1 \rightarrow O(K \log K)$ time
 - $H.\text{insert}((i, L[i][0]))$
- 5.) while ($H.\text{size}() \neq 1$) \rightarrow executed $N * K$ at most, therefore $O(N * K)$
 - Let $(i, v) = H.\text{extractMin}() \rightarrow O(\log K)$
 - add v to back of $A \rightarrow O(1)$
 - $A'[i] += 1 \rightarrow O(1)$
 - If ($A'[i] \neq N$)
 - $H.\text{insert}((i, L[i][A'[i]])) \rightarrow O(\log K)$
- 6.) let $(i, v) = H.\text{extractMin}()$
- 7.) for $j = A'[i]$ to $N-1$ // for last array simply add everything remaining to A
 - add $L[i][j]$ to back of A

Total time complexity = time complexity of step 5 = $O(N * K * \log K)$

8. Number Candies [15 marks]

The current rage among children are number candies. As the name suggest, these candies are given a number from 1 to N where N is at least 10 million and can be up to the **billions**! The ultimate goal is to collect all N of these candies. That is of course quite impossible since the number candies carried by any candy store is random (and with lots of repeats), and no store can carry anywhere near even 10 million candies.

However, the candy manufacturer has made the task easier. If anyone can collect all candies numbered sequentially from 1 to M where $M < N$ and up to 100,000, the manufacturer will give him/her all the N numbered candies.

One particularly rich kid has hired L helpers, where $M < L < N$ and up to 1,000,000. Each helper will buy one candy from each of the L candy store around the country. He hopes that by doing so he will be able to get all the candies numbered from 1 to M .

As his helpers come back one by one with the candy they have bought, the rich kid has a hard time trying to figure out if he has all the candies numbered from 1 to M .

Using what you have learned in CS2040, give an algorithm and required data structure(s) to help the rich kid answer the following query in better than $O(\text{Min}(\log N, M))$ time:

What is the largest numbered candy he has that is in an unbroken sequence from candy #1? If the largest numbered candy found in such a way is $> M$ just return M . If he does not have candy #1 then return -1.

You have to answer this query each time a candy is being brought back by a helper. So there will be L such queries (each to be answered in better than $O(\text{Min}(\log N, M))$ time). *Note that the candies need not be brought back in ascending order of candy number.*

E.g if a helper brings back candy #15 and the candies the rich kid has is currently 1,2,3,4,5,6,7,20,21,23,25,26,27,30, then after the inclusion of candy #15, the largest numbered candy he has that is in an unbroken sequence from candy #1 is candy #7.

You can perform a preprocessing step to help you answer the query. The preprocessing step must run in time better than $O(N \log N)$ time.

Please analyze the time complexity of your solution.

Since preprocessing must be better than $O(N \log N)$ time and each query must be answered in better than $O(\log N)$ time we cannot have a solution using AVL to store all N candies.

Algorithm 1: Preprocessing $O(M \log M)$ + answering query $O(\log M)$ time

In this solution instead of storing what candies the rich kid has, store what he does not have!

Preprocessing: Create an AVL tree T containing the numbers 1 to M . $\rightarrow O(M \log M)$

Answer query given candy j returned by helper:

```

if (T.search(j) != null)
    T.delete(j)
if (T.findMin() == 1) // still don't have candy 1
    return -1
else
    return min(100000, T.findMin()-1) // rich kid has every candy from 1 to (smallest value in T)-1

```

alternative answer 1:

Amortized $O(1)$ per query solution, no preprocessing, $O(L)$ space

Since we are only interested in a consecutive sequence of candies starting from candy 1, after each candy is added, we try to 'extend' the longest consecutive sequence starting from 1 by as much as we can. We can use a hash table / array to maintain the current collection of candies we have, so that we can check whether we have a certain candy quickly when we try to extend our consecutive sequence. The pseudocode is shown below:

```

Let H be a hash table, initially empty
Let A[i] denote the candy added in the ith query
cur_max = 0

for i = 1 to L
    H.insert(A[i])
    while H.contains(cur_max + 1)
        cur_max += 1
    if cur_max == 0
        print -1
    else
        print cur_max

```

The for-loop runs for $O(L)$ iterations. Note that over the $O(L)$ iterations of the for loop, the while-loop runs for $O(L)$ iterations in total, because if L candies are added to our collection, the longest sequence of consecutive candies that we can obtain starting from candy 1 cannot end further than candy L . Since the while-loop runs for $O(L)$ iterations in total across $O(L)$ iterations of the for-loop, each query can be answered in $O(1)$ time on average.

$O(1)$ per query solution, no preprocessing, $O(L)$ space.

Suppose that currently, we have the following candies in our collection:

1 2 3 4 5 6 7 10 11 12 13 14 25 26 27 114 115 116

We can treat the candies as a collection of intervals, where a sequence of consecutive candies form a single interval. For example, in the collection above, we have the interval $[1, 7]$ (since we have all the candies from 1 to 7), $[10, 14]$, $[25, 27]$, $[114, 116]$.

Whenever we get a query, a new candy is added to our collection. When a new candy is added to our collection, there are four different cases that might occur:

1. The new candy can be 'merged' into an interval on its right
For example, suppose that currently, we have the candies 5 6 7 8 in our collection, which means we have the interval [5, 8]. If candy 4 is added, we can 'merge' this candy into the interval to its right starting with 5, to obtain a new interval [4, 8].
2. The new candy can be 'merged' into an interval to its left
For example, suppose that currently, we have the candies 2 3 4 5 in our collection, which means that we have the interval [2, 5]. If candy 6 is added, we can 'merge' this candy into the interval on its left ending with 5, to obtain a new interval [2, 6].
3. The new candy can be 'merged' into both the intervals on its left and right
For example, suppose that currently, we have the candies 1 2 3 5 6 7 in our collection, which means that we have the intervals [1, 3] and [5, 7]. If candy 4 is added, we can 'merge' this candy into the interval on its left ending with 3 and the interval to its right starting with 5, to obtain a new interval [1, 7].
4. The new candy cannot be merged with any of the existing intervals
For example, suppose that currently, we have the candies 2 3 4 5 6 10 11 12 in our collection, which means that we have the intervals [2, 6] and [10, 12]. If candy 8 is added, we cannot merge this candy with any of the existing intervals, so we will create a new singleton interval [8, 8]. This means that after candy 8 is added, we have three intervals [2, 6], [8, 8] and [10, 12].

So, we are interested in maintaining a collection of intervals, and for every query where a new candy is added to our collection, we will either add a new interval to our collection of intervals, or merge this candy with either one or two of the existing intervals as shown in the cases above. This can be done efficiently using a hash table (or actually, even an array) in $O(1)$ time.

The hash table solution will be described here. The array solution is analogous.

We maintain three hash tables. The first hash table maps the start of an interval to the end of an interval, so it contains (start of interval \Rightarrow end of interval) key-value pairs. The second hash table maps the end of an interval to the start of an interval, so it contains (end of interval \Rightarrow start of interval) key-value pairs. The third hash table helps us to ignore duplicate candies. For each query, when a new candy is added, we try to merge this new candy with an existing interval according to one of the four cases above by manipulating the entries in the two hash tables.

To output the answer to a query, we output the end of the interval that starts with candy 1, or -1 if such an interval does not exist.

The pseudocode is shown below:

```

Let H1 be a hash table containing (start of interval  $\Rightarrow$  end of interval) key-value pairs
Let H2 be a hash table containing (end of interval  $\Rightarrow$  start of interval) key-value pairs
Let H3 be a hash table containing integers
Let A[i] denote the candy added in the ith query

for i = 1 to L
    if H3.contains(A[i]) // duplicate candy
        continue
    if H1.contains(A[i] + 1) and !H2.contains(A[i] - 1) // Case 1
        // First obtain the interval that we are going to merge with
        start = A[i] + 1
        end = H1.get(A[i] + 1)
        // Delete this interval from both H1 and H2
        H1.remove(start)
        H2.remove(end)
        // Add the new interval into both H1 and H2
        H1.add(A[i], end)
        H2.add(end, A[i])
    else if H2.contains(A[i] - 1) and !H1.contains(A[i] + 1) // Case 2
        // First obtain the interval that we are going to merge with
        end = A[i] - 1
        start = H2.get(A[i] - 1)
        // Delete this interval from both H1 and H2
        H1.remove(start)
        H2.remove(end)
        // Add the new interval into both H1 and H2
        H1.add(start, A[i])
        H2.add(A[i], start)
    else if H1.contains(A[i] + 1) and H2.contains(A[i] - 1) // Case 3
        // First obtain the two intervals that we are going to merge with
        start1 = A[i] + 1
        end1 = H1.get(A[i] + 1)
        end2 = A[i] - 1
        start2 = H2.get(A[i] - 1)
        // Delete both intervals from both H1 and H2
        H1.remove(start1)
        H1.remove(start2)
        H2.remove(end1)
        H2.remove(end2)
        // Add the new interval into both H1 and H2
        H1.add(start2, end1)
        H2.add(end1, start2)
    else // Case 4
        // Add a singleton range
        H1.add(A[i], A[i])
        H2.add(A[i], A[i])

    // Insert into H3 for tracking duplicate candies
    H3.insert(A[i])

    // Answer the query
    if H1.contains(1)
        print H1.get(1)
    else
        print -1

```

For every query, we do a constant number of insertions, finds and deletions to a hash table. Hence, every query can be answered in $O(1)$ time

alternative answer 2:

$O(\alpha(L))$ per query solution, $O(L)$ space

Suppose that currently, we have the following candies in our collection:

1 2 3 4 5 6 7 10 11 12 13 14 25 26 27 114 115 116

Let every consecutive sequence form a *set* of candies. So in our collection above, there are four sets of candies: {1,2,3,4,5,6,7}, {10, 11, 12, 13, 14}, {25, 26, 27}, {114, 115, 116}.

At each query, a new candy is added to our collection. This may cause two sets to be merged together, if this new candy links up two initially disjoint consecutive sequences. For example, if we have the following candies in our collection initially: 3 4 5 6 8 9 10 11, with two sets {3, 4, 5, 6} and {8, 9, 10, 11}, and if candy 7 is added, then these two sets will be merged into one combined set {3, 4, 5, 6, 7, 8, 9, 10, 11}.

After every query, we want to report the length of the longest consecutive sequence of candies starting from candy 1. Observe that this is effectively that largest candy in the same set as candy 1. Note that after two sets are merged together (as described in the paragraph above), the length of the longest consecutive sequence of candies starting from candy 1 cannot decrease. So, we will try to maintain the maximum candy number for every set, and update this maximum when two sets are merged.

This suggests a solution using UFDS. Initially, we place every candy in a separate set. When we receive a new candy, we try to merge it with the set containing the candy on its left and right, if the sets exist, and update the value of the maximum candy in the set. After every query, we output the value of the largest candy in the same set as candy 1. The pseudocode is shown below. Modifications to the UFDS data structure are shown in blue.

```

Let par[1..L] be an array of integers (parent array for UFDS)
Let rank[1..L] be an array of integers (rank array for UFDS)
Let max_candy[1..L] be an array of integers (we maintain the candy
with the largest value in each set here)
Let has_candy[1..L] be an array of booleans (this is to track if we already
have a certain candy)
Let A[i] denote the candy added in the ith query

function findSet(x)
    if par[x] == x
        return x
    return par[x] = findset(par[x])

function sameSet(x, y)
    return findSet(x) == findSet(y)

function mergeSet(x, y)
    x = findSet(x)
    y = findSet(y)
    if x == y
        return
    if rank[x] > rank[y]
        par[y] = x
        max_candy[x] = max(max_candy[x], max_candy[y])
    else
        par[x] = y
        if rank[x] == rank[y]
            rank[y]++
        max_candy[y] = max(max_candy[x], max_candy[y])

// UFDS initialisation
for i = 1 to L
    par[i] = i
    rank[i] = 0
    max_candy[i] = i
    has_candy[i] = false

for i = 1 to L
    if A[i] > L
        // if a candy has value higher than L, then we just ignore it
        // since it is impossible to get a consecutive sequence of candies
        // higher than L
        continue
    if has_candy[A[i]]
        // duplicate candy
        continue
    has_candy[A[i]] = true
    // some array out of bounds checking here omitted
    // since they're not essential to the solution
    if has_candy[A[i-1]]
        mergeSet(A[i], A[i-1])
    if has_candy[A[i+1]]
        mergeSet(A[i], A[i+1])

    if has_candy[1]
        print max_candy[findSet(1)]
    else
        print -1

```

For each of the L candies, a constant number of UFDS operations are performed, so each query is answered in $O(\alpha(L))$ time. The UFDS takes $O(L)$ space.

9. Time Traveler [25 marks]

Mark is an officer from the Bureau of Time Traveling Law Enforcement (BOTTLE for short). He has a device which allows him to go forward in time in order to apprehend time traveling criminals. Starting from the present day which we will call day 0, Mark can use his device to jump forward in time by a certain number of days.

Each time he uses the device he can choose a value from a set L ($|L| > 0$) containing the number of days to jump. For example, given $L = \{2, 5, 7, 10, 30\}$, Mark can choose 7 which means he will jump forward by 7 days into the future. He can make multiple jumps by doing so, however there is a maximum range of the number of days he can go forward to which is given by the value $T - 1$. Once he goes past $T - 1$, he will simply wrap around to day 0 and continue from there. For example, given L as above, if $T = 20$, and he chooses 30, after he goes past day 20 in the future he will simply return to day 0 and continue up to day 10 in the future which is where he will find himself.

Now choosing each value from L to make a jump requires a certain amount of chrono-energy and each value in L has an associated amount of chrono-energy which is represented in another set M . For example, again given $L = \{2, 5, 7, 10, 30\}$, and $M = \{4, 7, 10, 20, 40\}$, if Marks chooses 5 then he will need to use 7 chrono-energy to jump forward 5 days. If he chooses 10 then he will need to use 20 chrono-energy to jump forward 10 days. However, the time machine can only store X units of chrono-energy thus Mark can only make a limited number of jumps before he has to stop.

Given the set L , the set M , the maximum range T , the value F which is number of day in the future he wants to arrive at from current day (day 0) and X the amount of chrono-energy the time machine starts with, model the problem as a graph problem and answer the query **JumpPossible(L, M, T, F, X)** which will return true if he can make a series of jump to arrive at day F without running out of chrono-energy (he can hit 0 when he arrive at day F), or false otherwise.

For example, if $L = \{2, 5, 7, 10, 30\}$, $M = \{4, 7, 10, 20, 40\}$, $T = 20$:

Now if $F = 10$, $X = 8$, then there is no way for him to arrive 10 days in the future, since making 1 jump of 10 requires 20 chrono-energy, making 5 jumps of 2 require $5 \cdot 4 = 20$ chrono-energy, making 1 jump of 30 which will wrap around and land him at day 10 will require 40 chrono-energy. Any other combinations will also require too much chrono-energy.

Now if $F = 17$, $X = 31$ then one possible way for him to jump forward 17 days is by making the series of jumps: 7,10 which cost only 30 chrono-energy.

a) Model the graph $G(V, E)$ to answer **JumpPossible(L, M, T, F, X)**

i.) What do the vertices V represent and what information will you store in a vertex? [2 marks]

Vertices represent the number of days that Mark can jump to in the future. There is a vertex for each number from 0 to $T-1$

ii.) What do the edges E represent and what information will you store in an edge? [2 marks]

Edges are directed and for any vertex pair x, y there is an edge from x to y if there is an i such that $y = (x + L[i]) \% T$. The weight of the edge is $M[i]$. Ignore edges that point back to the vertex itself or multiple edges pointing to same vertex y (for these simply use the smallest weighted edge).

iii.) ~~Is this a directed or undirected graph?~~ [1 mark]

Actually it can have a combination of directed and undirected/bidirected (e. g if you can link vertex i to vertex $T-1$, meaning $T-1-i$ is in L then $(T-1+(T-1-i)) \% T$ will get you back i so there is also an edge from $T-1$ to i . **SO EVERYONE GETS 1 MARK FOR THIS QUESTION IF YOU GIVE AN ANSWER (DIRECTED OR UNDIRECTED).**

iv.) What is the best graph DS to store this graph to answer **JumpPossible(L,M,T,F,X)**? [1 mark]

Adjacency List.

b) Now using the graph in a) give an algorithm for **JumpPossible(L,M,T,F,X)** that runs in time better than $O(|L| * T^2)$. [10 marks]

Simply run original or modified dijkstra on the graph modelled in a) using vertex 0 as the source vertex. If $D[F] \neq -1$ && $D[F] \leq X$ then return true else return false

of vertices = T

of edges = $O(|L| * T)$

Thus time for dijkstra algo = $O((T + |L| * T) \log T) = O(|L| * T \log T)$ which is better than $O(|L| * T^2)$

c) Mark has modified his time machine so that in his series of time travel jumps to reach a certain day in the future, he can at **any point** and **only once** pick one day from a set L' of days ($L' \cap L = \emptyset$ and $|L'| \leq |L|$), that he can jump forward by without using any chrono-energy. Given this new modification, detail any changes to the graph in a) and give an algorithm for **JumpPossible(L,L',M,T,F,X)** that will still run in time better than $O(|L| * T^2)$. [9 marks]

For example, if $L = \{2, 5, 7, 10, 30\}$, $L' = \{1, 4, 8\}$, $M = \{4, 7, 10, 20, 40\}$, $T = 20$:

If $F = 12$, $X = 8$ then one possible way to jump 12 days into the future is to first make a jump of 2 for 4 chrono-energy followed by a free jump of 8, then followed by another jump of 2 for 4 chrono-energy to reach 12 days in the future. In all, this use exactly 8 chrono-energy. However, Mark cannot make a free jump of 4 followed by another free jump of 8 to reach 12 days, since he can only do a free jump once in his series of jumps.

1.) Make a copy of G call it G' → can be done in $O(|L| + T)$ time on adjacency list. Let the copy of a vertex x in G be x' in G' .

2.) For each vertex x in G, there is a directed edge to a vertex y in G' , if there is an i' such that $y = (x + L'[i']) \% T$. The weight of the edge is 0. → can be done in $O(T^2)$ time

3.) Thus we have a 2 layer graph consisting of G and G' with possible edges going from G to G' but not from G' to G.

4.) Now again run modified/original dijkstra's algorithm from vertex 0 in G.

If $(D[F] \neq -1 \ \&\& \ D[F] \leq X) \ || \ (D[F'] \neq -1 \ \&\& \ D[F'] \leq X)$ return true else return false

Time complexity is still $O(|L| \cdot T \log T)$ since # of vertices = $O(2 \cdot T) = O(T)$ and # of edges = $O(2|L|T + |L'|T) = O(|L|T)$

Alternative way:

Another way is to note that the free jump can actually be used anywhere because of the modulo (%) operation for example if a jump of 2 days, 2 day and 4 days will work (and 4 is the free jump) then 4,2,2 or 2,4,2 or any combination of the days will lead to the desired destination. Thus we can simply call modified Dijkstra on the graph in a) and check for all vertices $F' = F - k'$ where k' is in L' if there exist $D[F'] \leq X$. If there is then output true else output false.

~~~ END OF PAPER ~~~