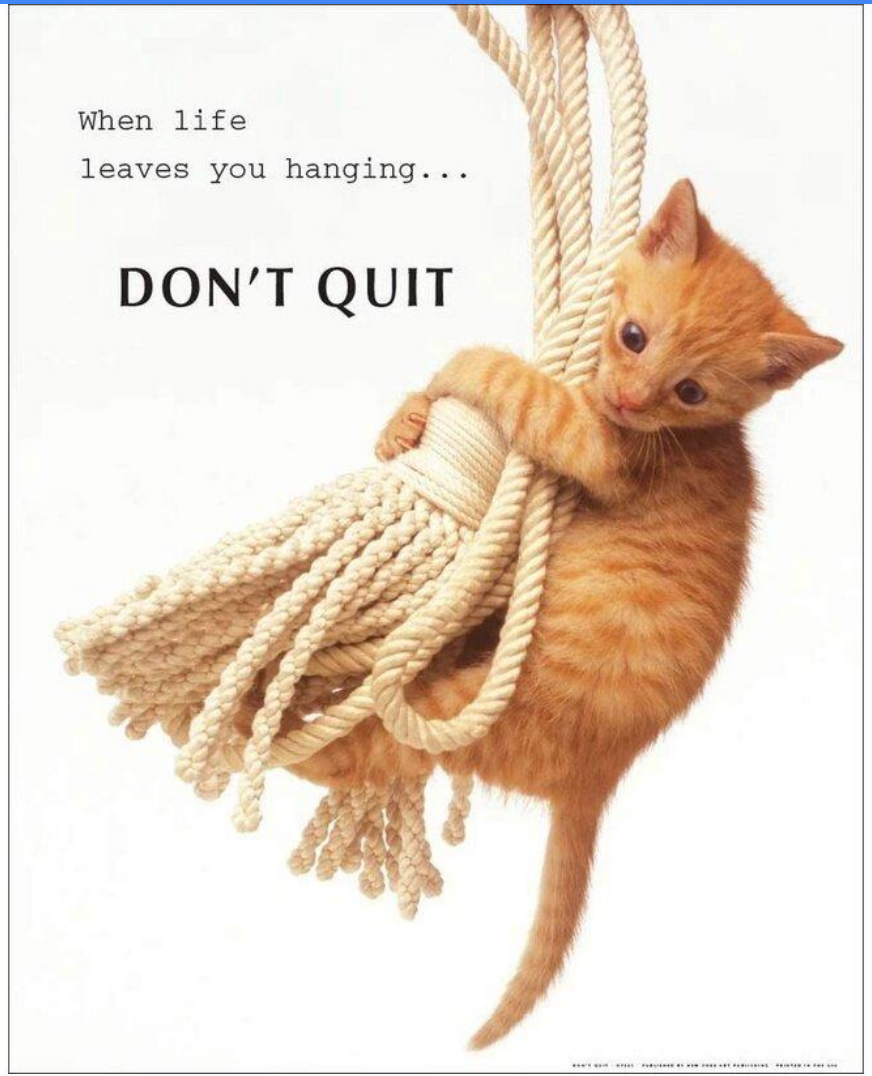
Chen Jun Hong
chen.junhong@u.nus.edu

CS2040 Lab 2

AY22/23 Sem 1, Week 4

Administrative Matters

- Unzipping folders
- Using check.sh
  - Ensure all files (.in, .out, .java, check.sh) are in the same folder
  - Run *bash ./check.sh* in that folder
- Private test cases? What are they?
- Reminder Visual algo starts next week!
- Lab Grading
  - Pre Condition, Post Condition, Comments, Variable Scoping, Modularity
  - Will grade the latest submission only
    - If there are multiple A submissions, we will only grade the contents of the latest one
    - If your latest submission is a B but you have an A submission, we will consider it a B grade submission
  - More details are on luminus: Files > Lab Materials > Lab Grading Criteria
- Overwhelmed?
  - Don’t worry! The start is always rough


When life
leaves you hanging...
DON'T QUIT

Common problems

- Single quotation marks must be used for Characters instead of double quotation marks
- Mistakenly thinking that io.getWord() takes in next Character instead of next String
- Forgetting to throw IOException when BufferedReader is used
- Indexing Error: Check for array indices to make sure it's still within limits
- Null Pointer Error: Referencing something that points to nothing in the memory
  - *trying to call null.charAt(), matrix[x][y].equals() when matrix[x][y] is null)*
- Code Indentation: A student prematurely closed Kattio because io.close() had wrong indentation
- How to traverse an array?
  - For loops:

```
for (int i = 0; i < lengthOfArray; i++) {
  someMethod(i);
}

for (arrayElementType element : array) {
  someMethod(i);
}
```

(to be continued….)

Common problems

- Failing only sometimes for the same question?
  - I/O slow causing TLE
  - Some loop take up to much time, caused by inefficient code and algorithm
  - Why does it happen? Codecrunch has different load at different time, so TLE may occur depend on the load at the time

- Public vs private? Will you be penalized?
  - Not CS2030s, not focused on OOP, you will not be penalized.
  - More focused data structures and algorithm
  - Modularity is still required, global variable should NOT be used

- Problem connecting to stu.comp.nus.edu.sg
  - stu points to stu1 by default
  - can use stu2 instead if stu1 is unavailable
  - If unable to connect, please let me know! (important for PE!)

Lab 1 - SpiralSnake

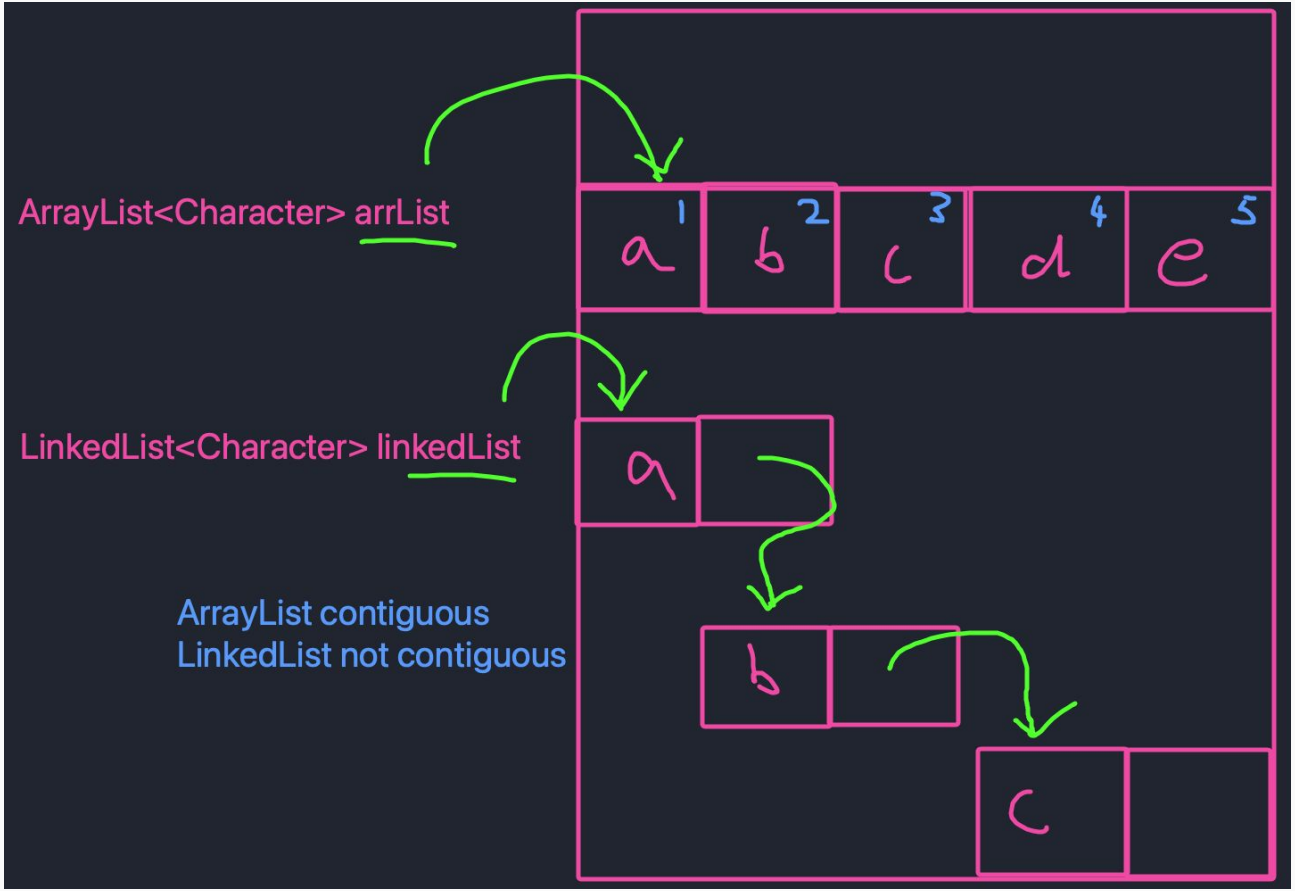
Having a 2D array MOVES will help greatly when traversing the array

- `int[][] MOVES = {{0,1}, {1,0}, {0,-1}, {-1,0}} //right, down, left, up directions`
- This might come in handy for future labs (when we reach graphs…) :)

Lab 2 - ArrayList vs LinkedList

	ArrayList Time Complexity	**LinkedList Time Complexity**
Insertion/Deletion to Front	O(N)	O(1)
Insertion/Deletion to Back	O(1)	O(1)
Retrieval (any index)	O(1) - due to *random access*	O(N)

Lab 2 - ArrayList vs LinkedList cont. (What is random access)



Lab 2 - Java API

Note that some methods provided by Java API may not run in O(1) time, and as such your program may run slower than intended if this is not taken into consideration
(Ex: LinkedList ‘addAll’ method)

In the following slides, I will go through some common/useful Java APIs to use when doing your lab assignments.

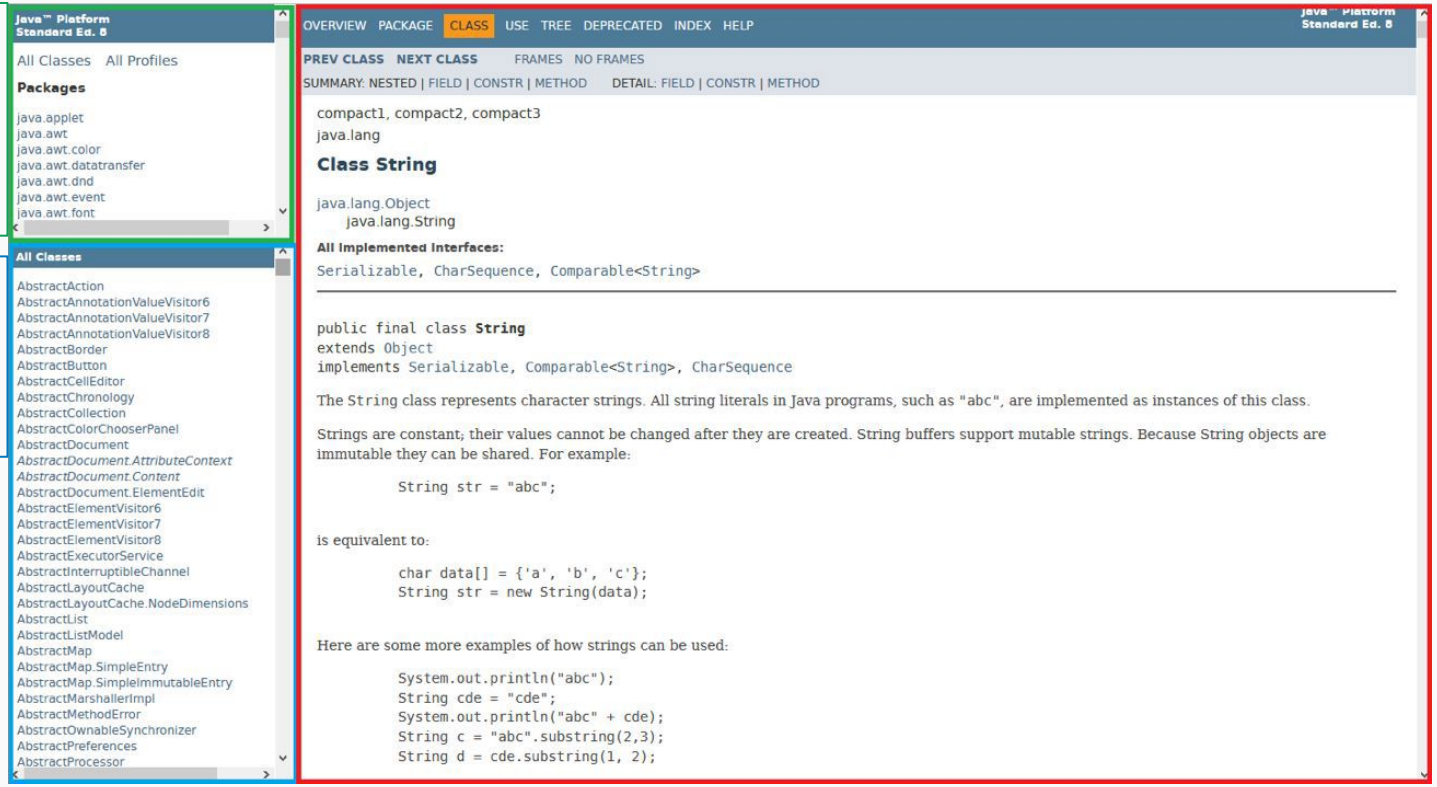
The slides are based off on the Java 8 API documentation: https://docs.oracle.com/javase/8/docs/api/

- Documentation of the later versions of Java (up to Java 10) follows the same idea, but uses a different layout by default. To view them in the same manner, click on “FRAMES” at the top of the page
- Documentation from Java 11 onwards do not seem to support frames, requiring manual navigation of the packages (or just search for the class name via Google, or the search bar at the upper right of the Java API page)

Lab 2 - Reading Java API

Package window: can click on them to view only classes from a single package

Class window: click on one of them to view the full API of that class



Main window: shows all the details of the selected class

Pro-Tip: Just do a quick google search (Data Struct) Java API

Lab 2 - Reading Java API

For the package window, clicking on a package will switch the class window to show only the classes in that package

- Frequently used packages are:
  - java.util
    - Almost everything that’s covered later in the module
  - java.lang
    - Strings
    - Wrapper classes
  - java.io
    - Buffered input/output

For Java 9 onwards, these are found under the java.base package on the documentation (but retain the same package names otherwise)

Lab 2 - Generics

- Most Java API usage from this point on uses Generics (denoted by arrow parentheses ‘<’ ‘>’)
- Essentially its a way to define the data structure/type being used, much like arrays
- Ex: String[] vs ArrayList<String>
- Can also be used for your own custom classes
- Ex: Deck[] vs ArrayList<Deck>
- Note: Primitives (int, long, float, double, boolean, char, etc) CANNOT be used as Generics. You will need to use their Wrapper classes (Integer, Long, Double, Boolean, Character, etc) instead.
- Ex: ArrayList<Integer>, LinkedList<Character>

Lab 2 - LinkedList API

Method	Return Type	Description	Time Complexity
.size()	int	returns the number of elements in the list	O(1)
.addFirst() .addLast()	E (Generics)	inserts specified element at the beginning/end of list	O(1)
.removeFirst() .removeLast()	E (Generics)	Removes and return the first/last element of the list	O(1)
.toString()	String	Inherits from AbstractCollection interface Returns the list in the format [e1, e2, e3..]	O(n)

Lab 2 - Iterating through a List

```java
/**
 * Driver method for program
 */
public void run() {
    LinkedList<Integer> linkedList = new LinkedList<>();
    for (int i = 1; i <=5; i++) {
        linkedList.addLast(i);
    }

    //Iterating through a list
    for (Integer item : linkedList) {
        System.out.println(item);
    }

    //Iterating through a list (iterator)
    ListIterator current = linkedList.listIterator();
    while (current.hasNext()) {
        System.out.println(current.next());
    }
}
```

Lab #2A

RhythmGame

RhythmGame - Problem Statement

- Total of n turns with combo building towards the right *by default*
- Each turn is represented by an integer ranging from 0 to 6
- 1, 2, 3, 4 are *normal moves* to be added to the current combo,
  5, 6 are special game play modes
- 5 *switches where moves are added* to the combo (From right to left, or vice versa)
  6 activates *alternating mode* where moves will be added to alternating sides (starting with the opposite side) and the next 6 deactivates alternating mode
- 0 denotes a *combo miss* and will reset the current combo back to an empty combo.
  (Note 0 is also *considered a move*, and will *not* reset special game play modes 5 and 6)
  - If we are in alternating mode 0 still counts as a move and we should continue to alternate where subsequent combos will be added

- **Input:** Integer n in a single line ($1 \le n \le 10^6$), followed by n integers ranging from 0 to 6 denoting each turn
- **Output:** Integer C in a single line denoting the maximum combo length achieved. Then in a new line, output the first sequence of moves that achieves this **maximum combo length.** (not the final combo!)

RhythmGame - Considerations

- $1 \leq n \leq 10^6$: Each move should be handled in O(1) time
- We need to keep track of the max combo while still updating/simulating the existing combo
  - In other words, we need to make and keep a copy of the existing combo if we deem it to be better but copying takes O(n) time
  - Checking if a combo is better and updating after every move will take $O(n^2)$ time, which is really inefficient.
  - Instead, we should only do the checks when we break a combo (or if we no more moves)
- We only need to check *after* a combo breaks.

Rhythm Game - Maintaining Max Combo

Input:

1	2	3	0	1	2	3	4

currentCombo:

1

Notice even though current combo is longer than the max combo so far, we do not copy yet

maxCombo:

Rhythm Game - Maintaining Max Combo

Input:

1	2	3	0	1	2	3	4

currentCombo:

1	2

Notice even though current combo is longer than the max combo so far, we do not copy yet

maxCombo:

Rhythm Game - Maintaining Max Combo

Input:

1	2	3	0	1	2	3	4

currentCombo:

1	2	3

Notice even though current combo is longer than the max combo so far, we do not copy yet

maxCombo:

Rhythm Game - Maintaining Max Combo

Input:

1	2	3	0	1	2	3	4

currentCombo:

maxCombo:

1	2	3

Combo broken, since current combo is longer, we now copy [1,2,3] and clear current combo

Rhythm Game - Maintaining Max Combo

Input:

1	2	3	0	1	2	3	4

currentCombo:

1

maxCombo:

1	2	3

Rhythm Game - Maintaining Max Combo

Input:

1	2	3	0	1	2	3	4

currentCombo:

1	2

maxCombo:

1	2	3

Rhythm Game - Maintaining Max Combo

Input:

1	2	3	0	1	2	3	4

currentCombo:

1	2	3

maxCombo:

1	2	3

Rhythm Game - Maintaining Max Combo

Input:	1	2	3	0	1	2	3	4

currentCombo:	1	2	3	4

maxCombo:	1	2	3

Rhythm Game - Maintaining Max Combo

Input:

1	2	3	0	1	2	3	4

currentCombo:

1	2	3	4

No more moves and since current combo is longer, we copy [1,2,3,4]

maxCombo:

1	2	3	4

Rhythm Game - Maintaining Max Combo

- In the worst case, every element is only ever copied once, so the total time complexity after processing all n queries will still be O(n)

Approach 1 - Simulate with ArrayList

- Use 2 booleans addRight, isAlternating to represent the different game modes.
- addRight represents which direction we should be adding combos. Encountering a ‘5’ will flip this.
- isAlternating represents if we are in alternating mode. If true, we should flip addRight every turn. Encountering a ‘6’ will flip isAlternating.

- Store current combo in an ArrayList, and add moves to left/right of the ArrayList depending on addRight
- Every combo break (either encountering a ‘0’ or end of inputs) check and update maxCombo

Problem with Approach 1 - Time Complexity Analysis

- Storing combo in ArrayList is too slow!
- Note that we must be able to support O(1) insertions to both the front and end of our combo.
- ArrayList is only able to insert to the end in O(1) time.
- Inserting to the front of the ArrayList take O(n) time because we have to shift back all other elements.
  - In the worst case we will have $O(n^2)$ time complexity
  - e.g [5,1,1,1,1,...]

- What data structure supports O(1) insertions to both ends? LinkedList

Approach 2 - Simulate with LinkedList

- Use 2 booleans *addRight*, *isAlternating* to represent the different game modes.
- *addRight* represents which direction we should be adding combos. Encountering a ‘5’ will flip this.
- *isAlternating* represents if we are in alternating mode. If true, we should flip addRight every turn. Encountering a ‘6’ will flip isAlternating.

- Store current combo in an LinkedList, and add moves to left/right of the LinkedList depending on addRight
- Every combo break (either encountering a ‘0’ or end of inputs) check and update maxCombo

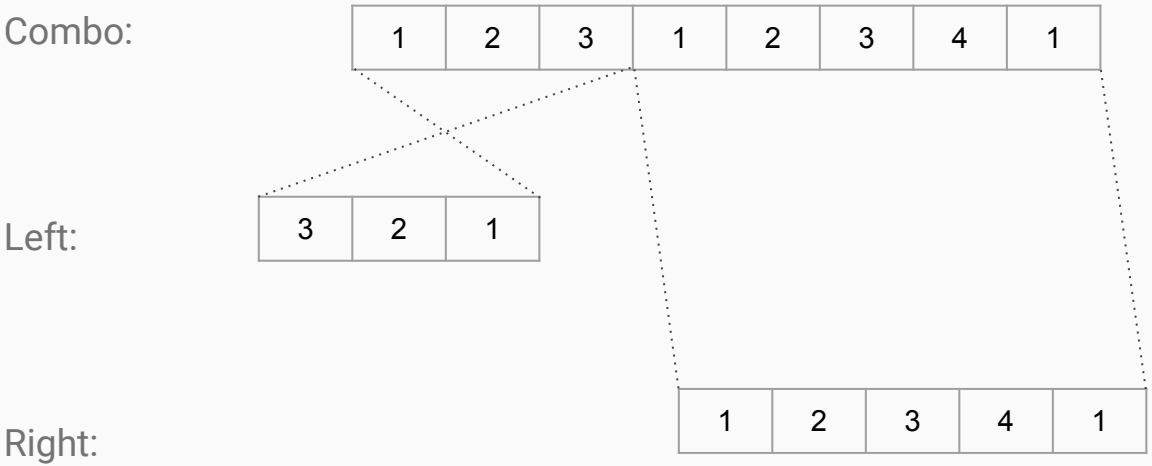
Approach 2 - Time Complexity Analysis

- Now insertions to both ends are supported in O(1) time.
- With n inputs, our algorithm now runs in O(n) time.

Alternative Method

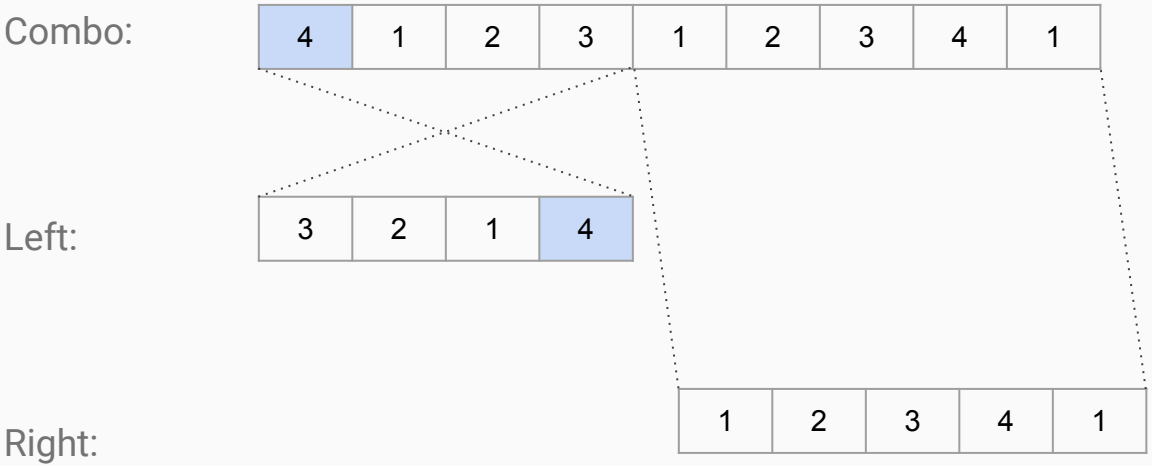
- Can we still use ArrayList to solve this problem efficiently? Yes!
- The only issue was insertions to the front of the combo.
- Instead of using 1 ArrayList to represent a combo, we use 2

Alternative Method



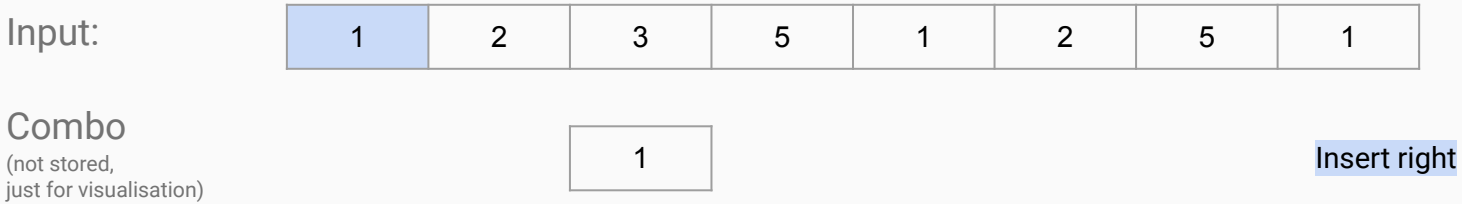
- Every combo represented by a left and right part. Right parts can handle insertions to the end in O(1) as per normal. Left part stores the left moves, but in reversed.

Alternative Method



- Now adding to the “front” of the combo corresponds to adding to the end of the left part. Front insertions have been converted to O(1) insertions to the end!

Alternative Method - Simulation



Alternative Method - Simulation

Input:

Combo
(not stored,
just for visualisation)

Insert right

Left:

Right:

Alternative Method - Simulation

Input:

Combo
(not stored,
just for visualisation)

Insert right

Left:

Right:

Alternative Method - Simulation

Input:

1	2	3	5	1	2	5	1

Combo
(not stored,
just for visualisation)

1	2	3

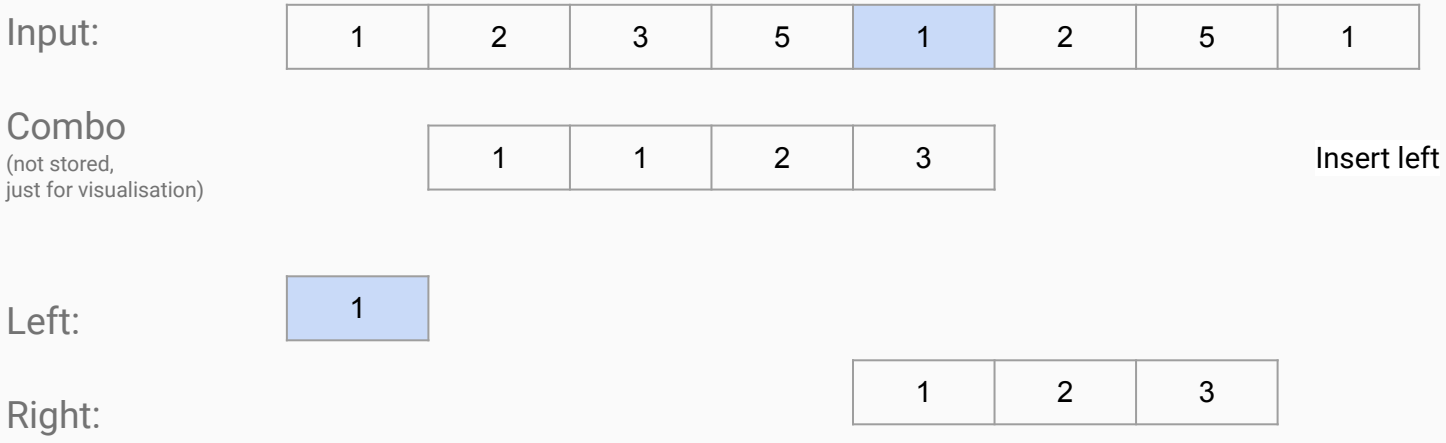
Insert left

Left:

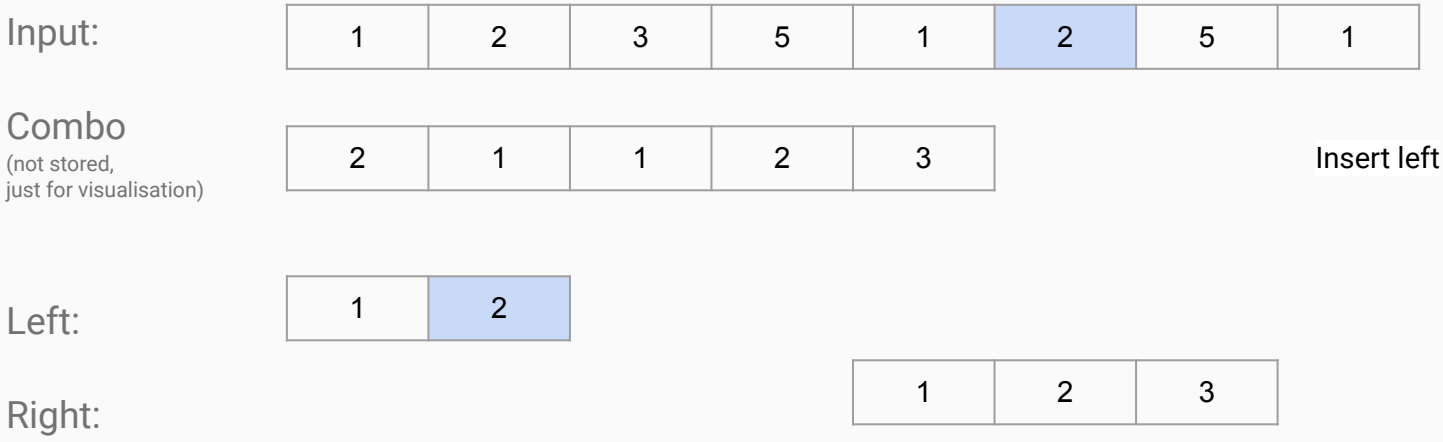
Right:

1	2	3

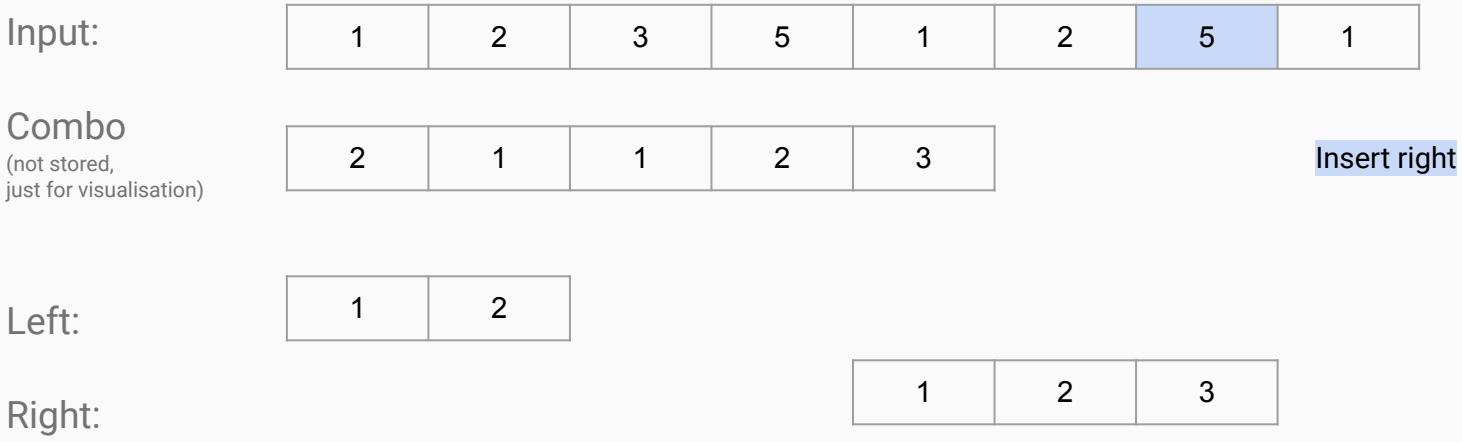
Alternative Method - Simulation



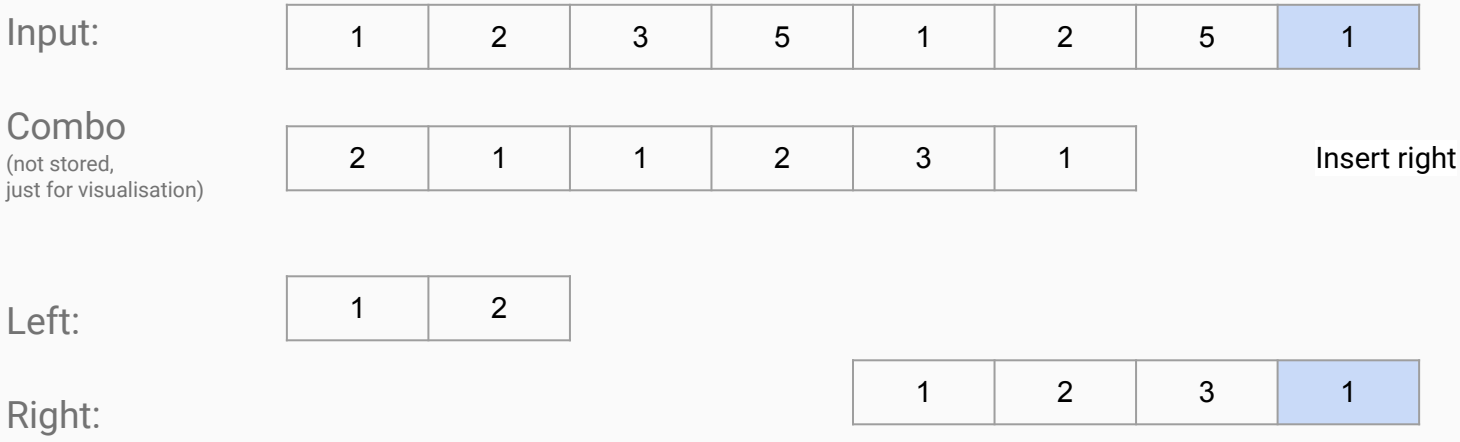
Alternative Method - Simulation



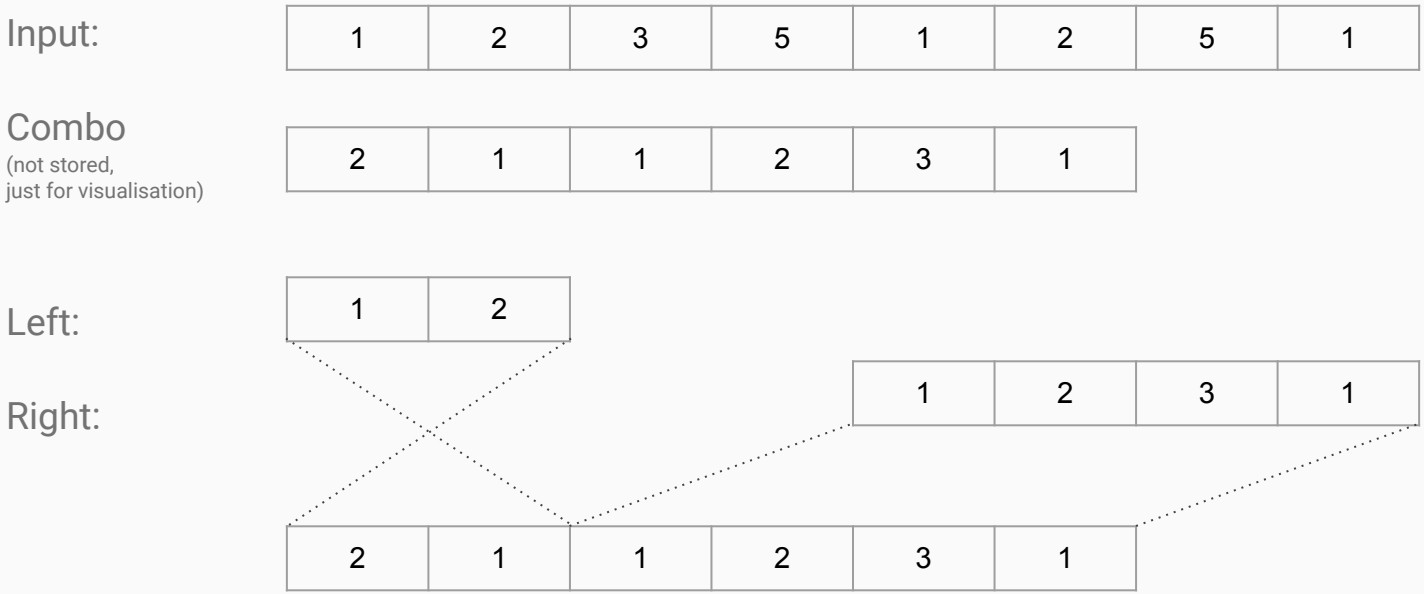
Alternative Method - Simulation



Alternative Method - Simulation



Alternative Method - Simulation



Alternative Method - Time Complexity Analysis

- Appending to front and back of the combo is simulated by an insert to the back of an ArrayList and so is supported in O(1) time.
- Thus n moves take O(n) time.

- Unlike the LinkedList implementation, this implementation also allows us to access the i-th combo move in O(1) time!

Lab #2B

Amusement Park - Graded

Problem Statement - Context

Problem Statement:

It's the summer vacation and you are taking your whimsical and indecisive little brother to a trip to Rocketland, a world famous theme park! Your goal is to determine for all days, which rides you are going to take with your little brother, judging from his various commands. Rides have ids in the range [1, 2^31 - 1].

Problem Statement - I/O

Input:

A list of commands each separated by a newline, with START RIDE being called once at the start of each day, and END being called at the end of all commands.

Output:

Print a single line, `"Invalid command"` for each invalid command.

For each day, print the final list of rides, in order, in this format: `"Day i: [a, b, … c]"` Each day's rides is on a single line.

Extract keywords

- Multiple days, each days have multiples rides
- Rides have id in range [1 - 2^{31}]
  - Overflow should not happen when using integer
- Parse commands to insert/remove rides, or switch days
- If command is invalid, print out **INVALID COMMAND**
- Output: Final list of rides for each days

Analyze - Executing commands

Potential threshold for efficiency (or why you are getting TLE):

- Processing of days
  - Usually not a problem, since after passing a day, no need to go back
- Processing of rides each day:
  - Can change a lot through the command, and need some efficient way to process
  - Ideally should be O(1) time complexity for all modifications of rides

Analyze - Modify rides

NEXT RIDE: A
DELETE FRONT RIDE: X
DELETE BACK RIDE: X
CHANGE FRONT RIDE: X A
CHANGE BACK RIDE: X A

- For each days, these are these are the commands that can change modify the number of rides
- No need to insert/remove random element in the middle, only need to do at the start and end of the list -> **Why ?**

Analyze - Modify rides

NEXT RIDE X

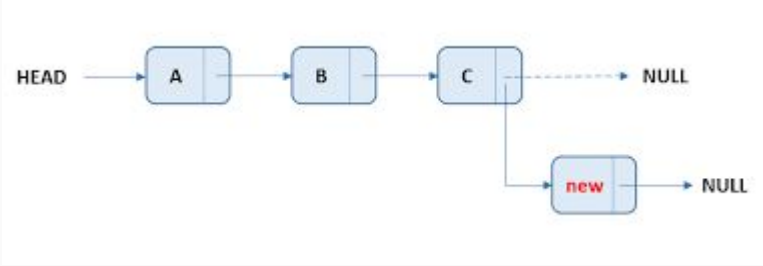
- Add new ride at the end of list

DELETE BACK RIDE X

- Remove ride at the end of the list X times

CHANGE BACK RIDE X A

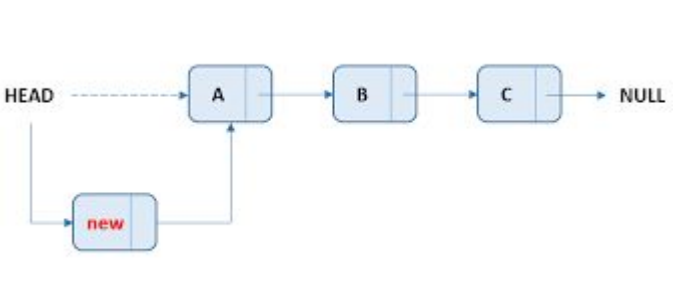
- Remove rides at the end X - 1 times, then change back ride



Analyze - Modify rides

Similarly, we have **DELETE FRONT RIDE X** , and **CHANGE FRONT RIDE X A**:

- Operate on the front instead of back



Approach

What data structure is suitable for:

- Quick insert and remove at the front of the list
- Quick insert and remove at the back of the list
- No need to have quick random element access?

-> Hint: LinkedList

Approach - Storing of the days

Consider what we need to do throughout the program:

- No predefined number of days
  - Cannot use Array
- Days will only passed, will not return back
  - No need to have random access
- Need to print out at the end
  - Days should be store sequentially

-> Hint: LinkedList

Any other data structure fit these requirement?

Alternative approach

Trying other data structures:

- Stack, Queue?
- ArrayDeque?

Your approach ?

For those who have attempted, failing for cases 50 onwards?

Lab #2C

HotPotato

Hot Potato – Problem Statement

- Total of n people, numbered 1 to n, stand around a circle.
- Starting from people 1, a hot potato is passed around in clockwise.
- Every k passes, the hot potato would “explode” and the person holding the hot potato would be eliminated, “disappear” from the game.
- This continues until there are only m people left in the game.
    - **Input:** In a single line, 3 integers, n, k, m

      (n = number of people ; k = number of passes for the potato to explode; m = number of people left in the end)

      Note that : $2 \le n \le 10^6$, $2 \le k \le 50$, $1 \le m \le 100$, $k \le n$, $m \le n$

    - **Output:** The final list of m people remaining in the game, in sorted order.

Illustration of the process (n = 6, k = 2, m = 2)

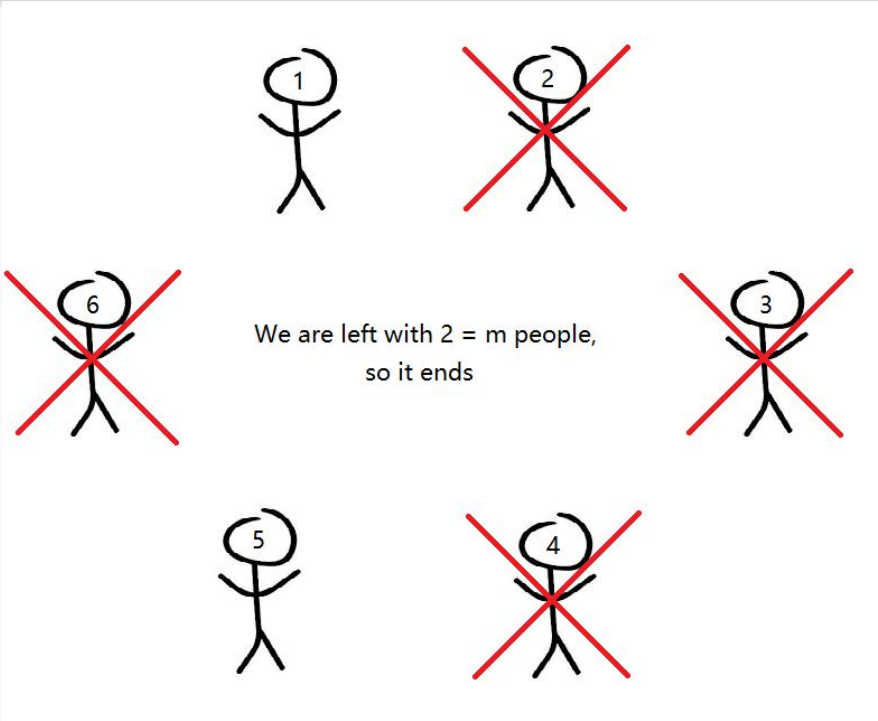


Illustration of the process (n = 41, k = 3, m = 1)

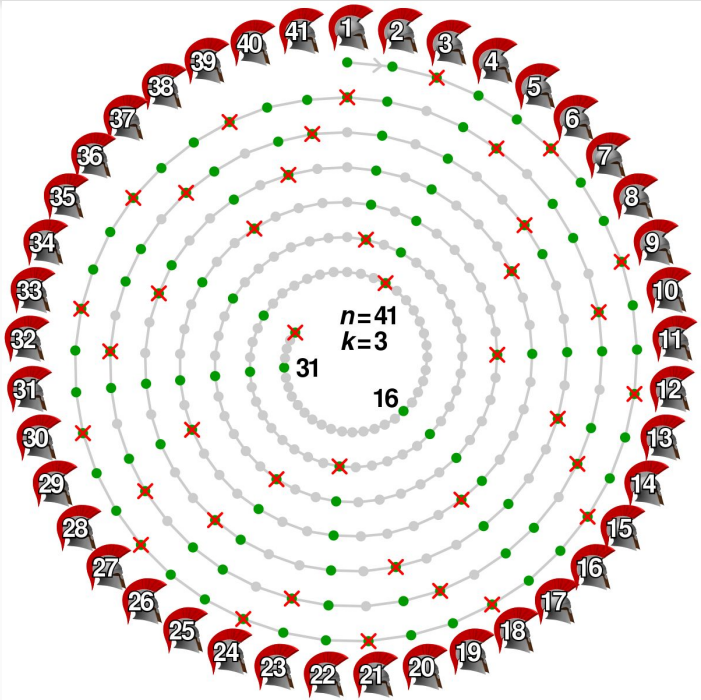


Photo from https://en.wikipedia.org/wiki/Josephus_problem#/media/File:Josephus_problem_41_3.svg

Approach – 1 Simulation with ArrayList

- We first try basic brute force simulation, with basic data structure array.
- Initialize an Arraylist with integer 1 to n inside, in order.
- While people left > m, at every step, do:
  - Countdown += 1 & current position += 1
    - if reach the end of array: start from index 0 again
  - When countdown reach k:
    - reset countdown back to 1
    - remove the people at current position, by index
    - People left -=1
- At the end, report whatever is left in the array

Keep record of:
- countdown of potato, int
- current position, int
- People left, int

Problem with approach 1 – Time Complexity Analysis

- Removing an element from the array costs $O(n)$, due to the shifting of the remaining elements
- We need to remove a total of n-m people
- Total time complexity of $O((n-m) * n) = O(n^2)$, since m is very small and insignificant when compared to m
- Then, we move k steps for each removal, costing O(nk), but it is ignored due to the $O(n^2)$ previously obtained

(to be continued….)

Problem with approach 1 – Time Complexity Analysis

- $O(n^2)$ is not good enough. How do we cut it down?
  - We still have to remove people one by one, so we can't cut down the $O(n - m)$ component.
  - Hence, we can reduce the cost for removing people.

- What kind of list structure allows faster element deletion?
  - LinkedList with cost $O(1)$
  - What type of linkedlist? Circular LinkedList

Approach 2: Simulation with LinkedList

- Similar idea to the Arraylist version:
- Initialize a Circular LinkedList with *1 to n* as value for each of the nodes
- While people left > m, at every step, do:
  - Countdown += 1 ; currentPosition = currentPosition.next.
    - Note that you *do not need to check* when reaching the end of list because now it is circular
  - When countdown reaches k:
    - reset countdown to 1
    - remove the node of current position
    - People left -=1
- At the end, report whatever is left in the LinkedList

Keep a record for:

- countdown of potato, int
- currentPosition, node reference
- People left, int

Approach 2 – Small subtlety

- When I remove a node, I have to link its previous node with its next node, how to deal with this:

- Method 1: [Code provided]
  - Maintain reference for current node, as well as the node before it

- Method 2:
  - Use doubly linked list

- Method 3: [Code provided]
  - Use ListIterator to do the removal and concatenation
  - Without the need to create your own node class!

Approach 2 – Alternative implementation using ListIterator

- Create a ListIterator for LinkedList, of values 1 to n.
- While people left > m, at every step, do:
  - Countdown += 1 & get next() from ListIterator.
    - When you reach the end of ListIterator: Restart from the start of your ListIterator
  - When countdown reach k:
    - reset countdown to 1
    - remove the node at current position
    - People left -=1
- At the end, report whatever is left
  - ListIterator can take care of this as well!

Keep a record for:

- countdown of potato, int
- current position, var, from ListIterator
- People left, int

Approach 2 – Time Complexity Analysis

- Now each removal only cost *O(1)*, as we only need to connect the previous and next node
- We need to remove a total of n-m people
- Total time complexity for deletion: *O((n-m)∗1) = O(n)* (since *m* is very small compared to *n*)

- For this approach, the moving cost become significant: We need to move *k* steps for each removal, with a total of *O(n)* removals, costing *O(nk)*.

Approach 3 – ArrayList Revisited

- Is it possible to use ArrayList for an efficient algorithm?
  - I cannot cut down the cost for deletion, so
  - For ArrayList, remove costs *O(n)* each, but addition only costs *O(1)*
  - Can I keep people left, instead of delete people eliminated?

- For each round around the circle
  - I create a new array with people survived that around
  - Use that new array for the next round

- Alternative explanation of where this insight come from
  - I revisit visited people only after completion of one round, so I can delay the deletion for these people until I finish one round
  - Kind of “lazy deletion”

Approach 3 – ArrayList Optimized

- While people left > m, at every step, do:
  - Countdown += 1 & current position +=1.
    - When you reach the end : Restart from the beginning using PeopleLeftForNextRound, empty PeopleLeftForNextRound
  - When countdown reach k:
    - reset countdown to 1
    - People left -= 1
  - If countdown is not k:
    - Add the current people to PeopleLeftForNextRound

- Keep a record for:
  - countdown of potato, int
  - current position, int
  - People left, int
  - An array of PeopleLeftForNextRound

- There might be people left in the unprocessed part of the original array, add them into PeopleLeftForNextRound
- At the end, report whatever is left using PeopleLeftForNextRound

Approach 3 – Time Complexity Analysis

- For each round, we do $O(n)$ work for collecting people one by one.
- After each round, the number of people left in the array decreased by $\frac{n}{k}$
- Let's write it down as a recurrence relation:
- $T(n) = T\left(n - \frac{n}{k}\right) + n = T\left(\left(\frac{k-1}{k}\right)n\right) + n$

Is there a way to think of this intuitively?

- Expand it further, $= T\left(\left(\left(\frac{k-1}{k}\right)^2\right)n\right) + \left(\frac{k-1}{k}\right)n + n$
- $= n + \left(\frac{k-1}{k}\right)n + \left(\left(\frac{k-1}{k}\right)^2\right)n + \left(\left(\frac{k-1}{k}\right)^3\right)n + \cdots + \mathrm{m}$
- This is (roughly) a geometric sequence, with $r = \frac{k-1}{k} < 1$, since m is very small compared to n
- Thus, $T(n) \approx \frac{u_1}{1-r} = \frac{n}{1-\frac{k-1}{k}} = O(nk)$. We get back the same time complexity again!

Some extensions

- This problem is a variant of a famous problem in computer science and mathematics, known as the Josephus Problem.
- Josephus Problem is a special case of this problem, when we set m = 1.
- There are well know solution for Josephus Problem with $O(n)$ and $O(klogn)$ time complexity.

- If you are interested, you can think about whether there exist a way to generalize that $O(n)$ and $O(klogn)$ solution to fit our problem.

Checking of SSH

Open up WinSSHTerm on computers

ssh into server
- hostname: stu.comp.nus.edu.sg
- username: e0123..
- password: nus account password

exit from server
- type ‘exit’ in the command line