CSE101

DUE NOV. 07

23:59 KST

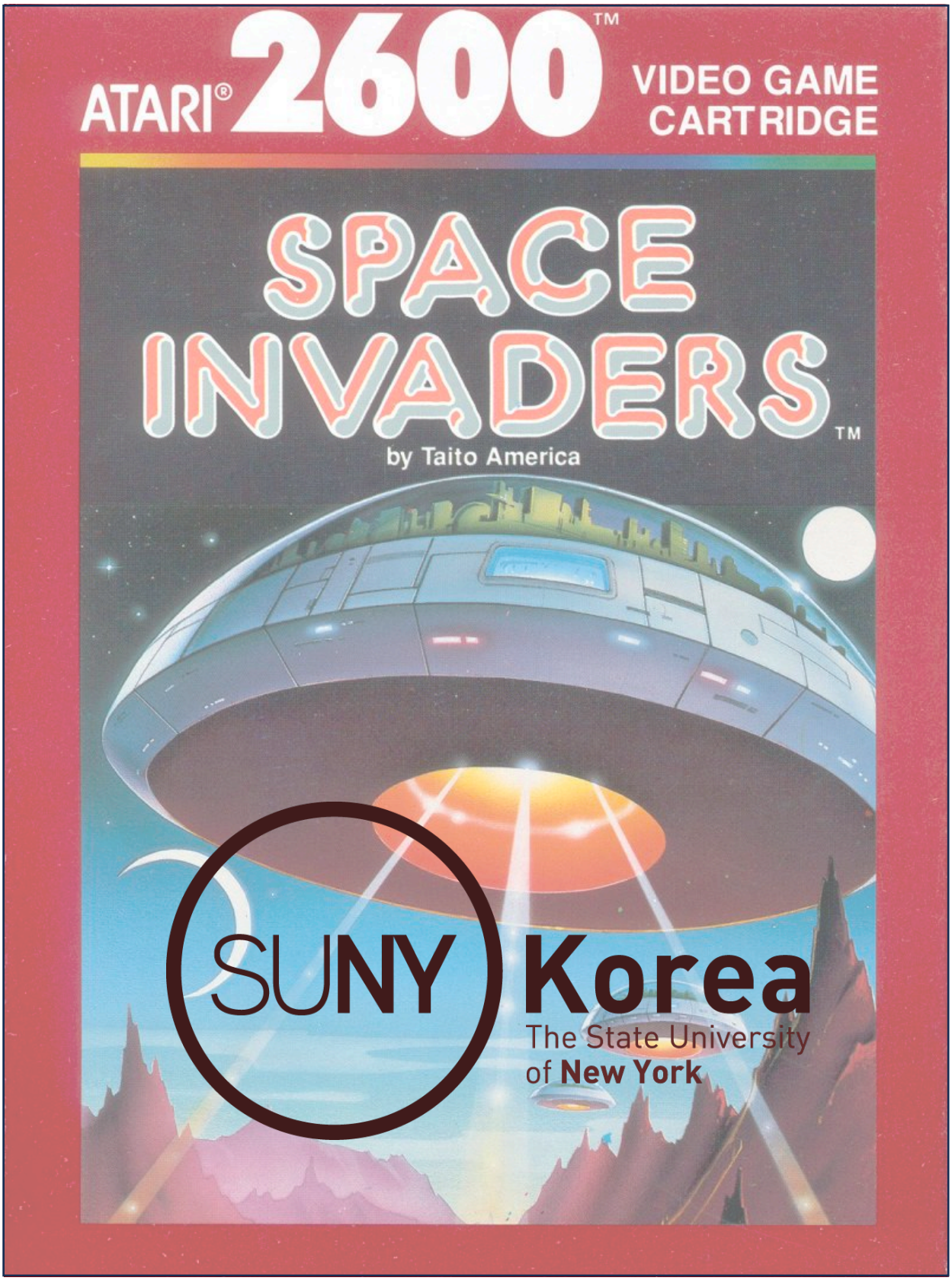
ASSIGNMENT III

PROFESSOR

FRANCOIS

RAMEAU

SEARCH



General instructions

- Please add your name and email in the dedicated location at the top of your code
- Do not use any external library except when explicitly requested
- Try to follow the naming convention proposed by PEP-8 seen in class
- Use ONLY what we have already seen in class. If not, you will get ZERO points
- Use meaningful names for your variables and functions
- If you face a problem submitting your code via GitHub, please contact the professor and the TA by email
- Note that the received code will be tested on a classifier to detect the potential usage of Large Language Models. We will also pay particular attention to plagiarism
- Leave comments in your code to explain your code and describe the difficulties you faced

INVITATION LINK

https://classroom.github.com/a/8f-JsOA2

In this homework, you will “pimp” a bit the binary search we have studied in class; instead of dividing the list into two equal parts, you will divide the list into three pieces and continue the search in the valid interval. This approach is called a **ternary search**.

Definition:

Ternary search is a divide-and-conquer search algorithm that divides a sorted list into three equal parts and determines which segment the queried element lies in, thereby narrowing down the search space to just that segment. Figure 1 illustrates this process.

Description:

In the ternary search, the list is divided into three parts by calculating two midpoints. If the desired element is equal to either of the midpoints, the position is returned, and you are done! Otherwise, based on the comparison with the midpoints, the search is narrowed down to one-third of the list:

1. If the desired element is less than the first midpoint, the search continues in the first segment.
2. If it's between the first and second midpoint, the search proceeds into the second segment.
3. If it's greater than the second midpoint, the search continues in the third segment.

This process repeats until the element is found or the search space is exhausted. Although ternary search performs fewer comparisons than binary search, it's not always faster due to the overhead of managing multiple segments and the extra comparisons in each iteration.

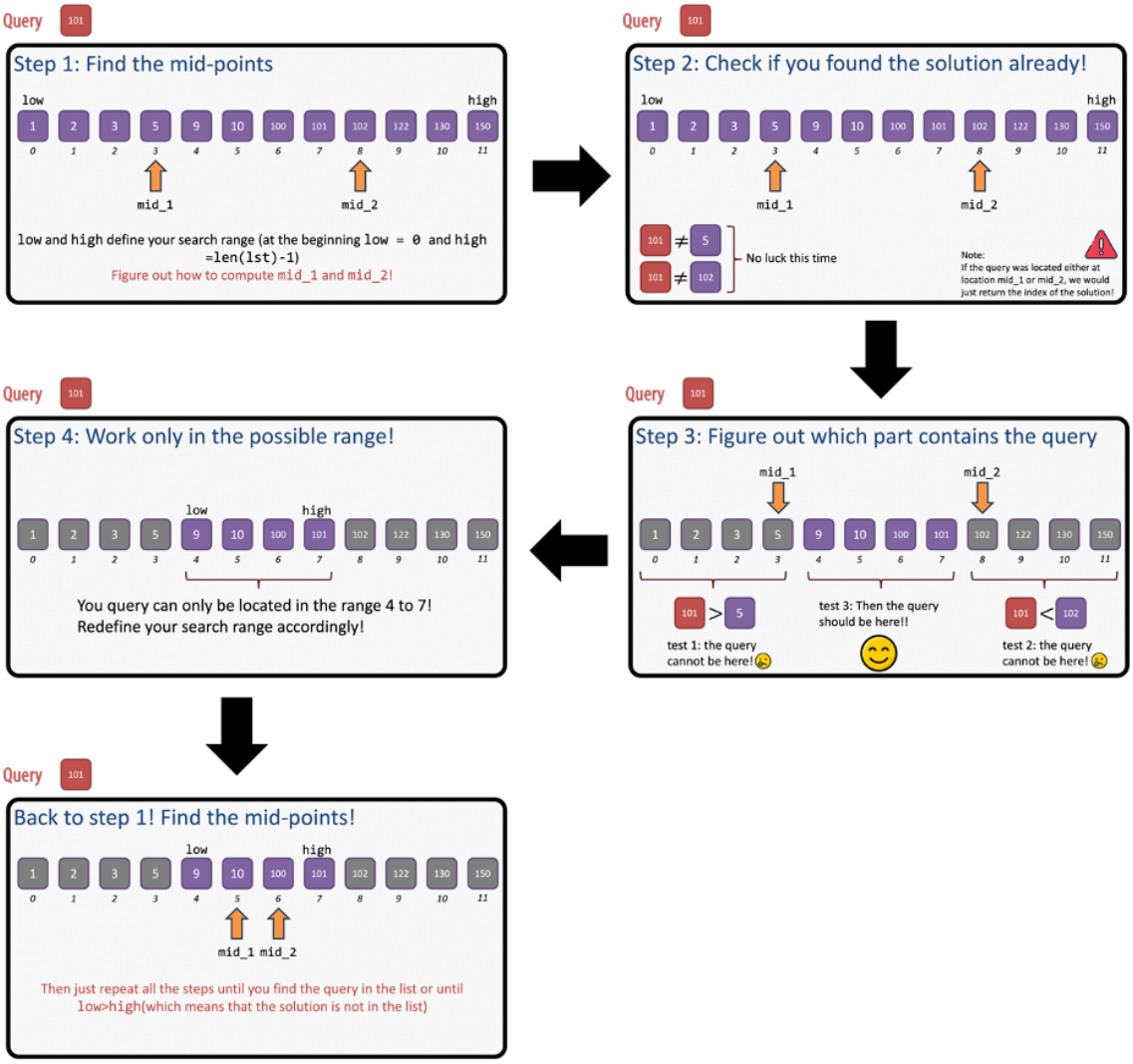


Figure 1 - Illustration of ternary search

To help you in your implementation, we will proceed in successive steps:

1. Determining if the query belongs in the range of the list
2. Implementation of the two mid-points calculation
3. Implementation of the ternary search algorithm
4. Using our implementation in a concrete scenario

Exercise 1: Check if the solution exists in the list (2 points)

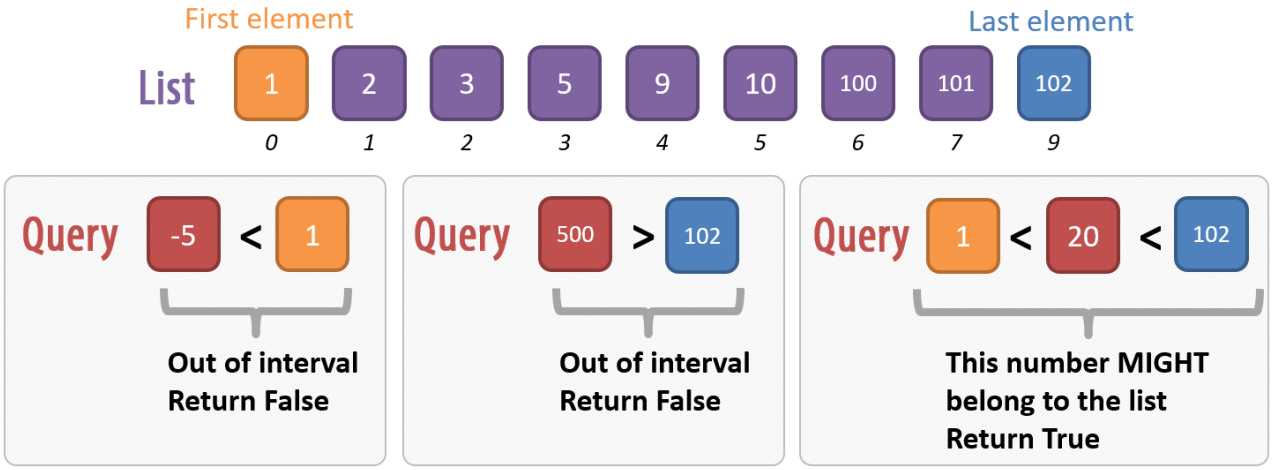
Since the ternary search algorithm only works on sorted lists, without looking at every single element, the first crucial step is to determine if the query item can even exist in the given list.

Objective:

To implement a function called `doesExist()`, which will take in the list and the query item as arguments. This function will serve as a preliminary check before using the ternary search. The function should quickly deduce if the query item lies within the range of the minimum and maximum values of the list.

Instructions

1. If the query item is smaller than the minimum value or greater than the maximum value of the list, return False, indicating the item doesn't exist. — 0.5 points
2. If the item lies within the range, return True, indicating that it's worth proceeding with the ternary search. — 0.5 points
3. Make sure your function works efficiently with a time complexity of O(1). — 1 point



Exercise 2: Compute the mid-points (2 points)

In the ternary search, it's essential to identify the two mid-points that divide the list into three almost equal parts. By doing so, we can then effectively reduce the search space by comparing the values at these mid-points with the target value.

Objective:

Implement a function named `computeMidpoints()`, which takes the current low and high indexes (defining the active range of the search) of your search space in the list and calculates the two mid-points.

Instructions:

Instructions

1.	The function should accept two parameters: `low` and `high` indexes in which to compute the midpoints	0 points
2.	Compute the first mid-point, `mid_1`, as: `low + (high - low) / 3`	0.5 points
3.	Compute the second mid-point, `mid_2`, as: `low + 2 * (high - low) / 3.`	0.5 points
4.	The function should return both mid-points: `mid_1` and `mid_2`.	0.5 points
5.	Ensure that the mid-points are integers!	0.5 points

Exercise 3: Implement the ternary search (3 points)

Ternary search is an advanced search algorithm that aims to find a target value within a sorted list by dividing the search interval into thirds and eliminating two-thirds of the search space based on comparisons.

Objective:

Implement a function named `ternarySearch()` which, given a sorted list and a target value, returns the index of the target value within the list if it exists, or -1 if the target value does not exist in the list.

Instructions

1.	The function should accept the following parameters: `sortedList`, `query`	0 point
2.	The first step is to check if the query can even exist in the given list, use your `doesexist()`function you implemented before. If the query cannot be in the list then return -1, else perform the search	0.5 point
3.	Implement the search algorithm as described at the beginning of this document. Do not forget to use your function `computeMidpoints()`	1.5 point
4.	If `low` exceeds `high`, the target value does not exist in the list. Return -1.	0.5 point
5.	Consider corner cases such as empty list	0.5 point

Exercise 4: Apply your ternary search (3 points)

Imagine an online video game where players earn points based on their performance in battles, quests, and other activities. These points determine their rank on the global leaderboard. With thousands of players active at any given moment and even more in the history of the game, it's crucial that the ranking system is efficient.

Objective:

When a player logs in, the game should immediately display their rank and tell them how many more points they need to surpass the player right above them.

Exercise:

Given a sorted list of player points (from highest to lowest), write a function that takes a player's points as input and returns their rank and the points needed to surpass the next player. Use our ternary search to achieve this. This code will be implemented in the **gameRanking()** function, this function takes as input the `scores_lst` and the `player_score` and return the `player_rank` and the gap (difference to rank up with respect to better player)

Instructions

1.	Use ternary search to find the player's current position (rank) in the sorted list. **Be careful, you might need to do some adjustment on the list to make it compatible with your implementation!**	1 point
2.	Calculate the difference in points between the found player and the player above them.	1 point
3.	Return the rank and points difference	1 point