Programming Assignment 1: Searching in 1-D Dynamic Data Structures

CS 5115 -Programming Prep for Grad

Instructor: Dr. Ajay K. Gupta

Western Michigan University

Submitted By: Afsana Sharmin

Problem Specification

I have to write a Python application to solve the following problem:

1) Use dataset Open English Word List (EOWL) - https://github.com/dwyl/english-words

2) Store the words in EOWL into a sorted dynamic-array which the program will manipulate. Start with size 2 dynamic array eowl[] and increase the size of eowl[] using a few different strategies when the current eowl[] array becomes full.

3) Increase strategyA: Incremental: increase the size of eowl[] by 10.

4) Increase strategyB: Doubling: double the size of eowl[].

5) Increase strategyC: Fib: use Fibonacci number sequence to increase the size of eowl[].

6) Use binary-search to insert the new word into eowl[], which will be done after allocating new array when an increase is needed.

7) Measure time at each search point.

8) Perform a theoretical complexity analysis of your design (i.e., count number of operations/instructions and space usage) and then express that using asymptotic notation as a function of the input size (Pause: what is the input size of your problem?)

9) Repeat steps 2 and 6 by simply using lists in Python which implement a version of ArrayLists.

10) Empirically measure the time and space complexity of your code (step 7 above should help towards that).

11) Compare the complexities from steps 8) and 10). (Long pause: in order to compare, what all you will need to do? We will discuss some alternatives in class, so stay tuned and this writeup may be modified accordingly at a later date.)

12) No need to print the whole eowl[]. At each increase point print the current size of eowl[], time elapsed and the following elements: 1st, [n/4]th, [n/2]th, [3n/4]th and nth, separated by à.

1. Introduction

This report covers the development process of a Python-based application designed to implement a custom Dynamic Array with different resizing strategies (Incremental, Doubling, Fibonacci) and analyze its performance in terms of time and space complexity. The project follows the Software Development Life Cycle (SDLC) phases to ensure systematic and organized development from conception to deployment and maintenance.

2. SDLC Phases

2.1. Requirements Analysis

The program needed to meet the following requirements:

Functional Requirements:

- Dynamic array class capable of handling incremental, doubling, and fibonacci resizing strategies.
- The ability to insert new elements and automatically resize the array when full.
- The capability to sort the array efficiently.
- Provide output regarding time and space complexity for both insertion and sorting and searching.
- Implement this code by python built-in list

Non-Functional Requirements:

- The code should be efficient and provide amortized time complexity insights.
- The system should be user-friendly, allowing easy selection of resizing strategies.
- The program must handle large datasets without crashing due to memory or performance issues.

2.2. Design

The design phase focused on defining the architecture and components of the system:

Modular Design: The program was broken into separate components:

Program 1: Custom Dynamic Array

a. Dynamic Array Class: Implements resizing strategies and related methods.
b. **File Handling**: Read text file for input data
c. Binary Search to get correct position for inserting new word
d. Main Program: To handle user interaction and select the desired resizing strategy.

Program 2: Dynamic Array with Python built in List

a. Initialize List
b. **File Handling**: Read text file for input data to store into list
c. Binary Search to get correct position for inserting new word
d. Main Program: To handle user interaction and get desired output.

2.2.1 Implementation

For Program 1: (CustomDynamicArray.py)

Imports:

- tkinter for file selection dialog.
- time for measuring execution time.
- sys for checking memory usage.

a) Dynamic Array Class:

class DynamicArray:

A custom dynamic array class that automatically resizes based on the chosen resizing strategy.

Attributes:

size (int): Current allocated size of the array.

array: Initialize with the given size

length (int): The number of valid elements in the array.

strategy (str): The resizing strategy ('incremental', 'doubling', or 'fibonacci').

fib1, fib2 (int): Values used for Fibonacci resizing strategy.

start_time (float): Time when the array was created to measure time elapsed during resizing.

def __init__(self, initial_size=2, strategy='any'):

Initialize the dynamic array with an initial size and a resizing strategy.

Args: initial_size (int, optional): The initial size of the array. Defaults to 2.

strategy (str): The resizing strategy ('incremental', 'doubling', or 'fibonacci').

def __getitem__(self, index): Get the item at the specified index.

Args: index (int): The index of the item to retrieve.

Returns: The value at the specified index

def __setitem__(self, index, value): Set the value at the specified index.

Args: index (int): The index where the value will be set.

value: The value to be set at the specified index.

def append(self, value): Append a new value to the end of the array, resizing if necessary.

Args: value: The value to append to the array.

def resize(self): Resize the array based on the selected strategy.

The size increases either incrementally, by doubling, or by the Fibonacci sequence.

def sort_array(self): Sort the dynamic array in ascending order and print the time and space taken to perform the sort.

def print_resize_status(self, time_elapsed):Print the details according to some positions after resizing, including the new size, time elapsed, and space used.

Args: time_elapsed (float): Time taken for resizing.

def __len__(self): Return the number of valid elements in the array.

def __str__(self): Return a string representation of the dynamic array.

b) File Handling: Enabled users to select a file of words for insertion into the array.

Library:

“import tkinter as tk from tkinter “

“import filedialog”

def select_file() : Open a file dialog to allow the user to select a file. It returns str that is the path to the selected file.

def read_file_into_dynamic_array(file_path, strategy) :

Read a file and store its words into a dynamic array.

Args: file_path (str): The path to the file. strategy (str): The resizing strategy for the dynamic array. Returns: DynamicArray: A dynamic array containing the words from the file.

c) Binary Search to get correct position for inserting new word

def binary_search(arr, target): Perform a binary search on the sorted array to find the target or the insertion point.

Args: arr (DynamicArray): The sorted dynamic array.

target (str): The target element to search for.

Returns:

Int: the index of the target if found, or the insertion point if not found.

def insert_element(arr, element): Insert an element into the sorted array and shift the elements accordingly.

Args: arr (DynamicArray): The sorted dynamic array.

element (str): The element to insert.

d) Main Program: To handle user interaction and select the desired resizing strategy.

__name__ == "__main__":

- Select input file
- Select resizing strategy
- Sort the array if needed
- Taking new word as input to insert the new allocated array
- Find the time and space needed for the program

For Program 2: (DynamicArrayList.py)

Imports:

- tkinter for file selection dialog.
- time for measuring execution time.
- sys for checking memory usage.

Functions:

i. read_file_into_list: Reads words from a file and appends them to a list.
ii. total_size_of_list: Calculates the total memory used by a list.
iii. binary_search: Performs binary search on a sorted list to find an insertion point.
iv. insert_element: Inserts a word in a sorted list using binary search.
v. sort_list: Sorts the list alphabetically and measures execution time and memory.
vi. select_file: Allows the user to select a file from the file dialog.

Main Logic:

- The user selects a file.
- The content of the file is stored in a list.
- Optionally, the list can be sorted.
- The user can interactively insert words into the list, maintaining order.

2.2.2: Complexity analysis

Theoretical complexity analysis of this program for large dataset

Key Operations & Their Complexity:

1.Initialization: An instance of the DynamicArray class, it initializes an array of a specified initial size.

- **Time Complexity**: O(m) where m is the initial size of the array. Here m=2
- **Space Complexity**: O(m) since you allocate space for m elements.

Setitem(), getitem(), len(), print() this kind of operation will take O(1) time

2.Appending Elements with resizing:

When appending elements, if the array is not full, adding an element takes O(1) time.

However, when the array is full, the resize operation is triggered, which copies all existing elements into a new array of larger size.

Depending on the resizing strategy, the amortized time complexity for multiple appends is affected differently:

Incremental Strategy: The array size is increased by a fixed amount (e.g., 10). Resizing happens after every 10 elements, leading to a time complexity of $O(n^2)$ due to repeated copying. It takes long time to generate dynamic array for large dataset.

Doubling Strategy: The array size doubles when full. This leads to an O(1) amortized time complexity for appending since the total number of resizing operations grows logarithmically.

Fibonacci Strategy: The size increases according to Fibonacci numbers. This leads to a complexity close to O(n), similar to the doubling strategy in the long run, but slightly worse due to the nature of Fibonacci growth.

3.Sorting the Array:

Sorting the array involves using Python's built-in sorting function (sorted), which has an average time complexity of O(n log n), where n is the number of valid elements in the array.

4.Binary Search:

Binary search on a sorted array takes O(log n) time, where n is the number of elements in the array.

5.Insertion in Sorted Array:

Inserting an element in the correct sorted position involves performing a binary search

(which is O(log n)), and then shifting elements to the right. The shifting operation takes O(n) time in the worst case.

Therefore, the complexity of inserting an element in a sorted array is O(n).

Space Complexity:

Dynamic Array Space:

The space complexity depends on the resizing strategy:

- Incremental Strategy: After appending n elements, the array size will be slightly larger than n Hence, the space complexity is O(n).
- Doubling Strategy: The array size is always a power of 2, so after appending n elements, the size will be approximately 2^k, where $2^k \geq$. Hence, the space complexity is O(n).
- Fibonacci Strategy: After appending n elements, the array size will be the closest Fibonacci number larger than n. Thus, the space complexity remains O(n).

Additional Space:

The only additional space used is for variables and the temporary array created during sorting or resizing, which do not asymptotically exceed O(n).

Analysis for Input Size n=370104

Incremental Strategy:

Appending all elements would require $O(n^2)$ operations, which would be computationally expensive and inefficient.

Doubling Strategy:

Appending all elements would require O(n) operations (amortized), which is significantly faster than the incremental approach.

Fibonacci Strategy:

Appending would still be O(n), although slightly slower than the doubling strategy but far more efficient than the incremental strategy.

Empirically measure the time and space complexity

Comparison time & Space:

Testcase	Incremental	Doubling	Fibonacci	Python List
Testcase1 =50 words	1.Dynamic Array Creation time =0.006180000 sec Space=472 bytes 2.Sorting time =0.000031200 sec Space =456 bytes 3.Insertion (backbit) Search time= 0.000039100 sec Space = 472 bytes	1.Dynamic Array Creation time =0.004372400sec Space=568bytes 2.Sorting time =0.000024800sec Space =456 bytes 3.Insertion (backbit) Search time= 0.000037600sec Space = 568bytes	1.Dynamic Array Creation time = 0.004486100 sec Space=496 bytes 2.Sorting time =0.000022400sec Space =456 bytes 3.Insertion (backbit) Search time= 0.000028500 sec Space = 496 bytes	1.Dynamic Array Creation time =0.000669900 sec Space=2959 bytes 2.Sorting time = 0.000272200 sec Space = 2959 bytes 3.Insertion (backbit) Search time= 0.000013100 sec Space = 3007 bytes
Testcase2 =100 words	1.Dynamic Array Creation time = 0.006628900 sec Space=872 bytes 2.Sorting time =0.000034500 sec Space =856 bytes 3.Insertion (aband) Search time= 0.000037700 sec Space = 872 bytes	1.Dynamic Array Creation time = 0.003393600sec Space=1080bytes 2.Sorting time =0.000042900 sec Space =856 bytes 3.Insertion (aband) Search time = 0.000031300 sec Space = 1080bytes	1.Dynamic Array Creation time = 0.006219900 sec Space=1208bytes 2.Sorting time =0.000023900sec Space =856 bytes 3.Insertion (aband) Search time = 0.000046800sec Space = 1208 bytes	1.Dynamic Array Creation time = 0.008363500 sec Space=5725 2.Sorting time =0.000369800 sec Space =5725 bytes 3.Insertion (aband) Search time = 0.000029900 sec Space = 5771 bytes

Testcase3 =602 words	1.Dynamic Array Creation time = 0.046144300 sec Space=4872bytes 2.Sorting time =0.000162800sec Space =4872bytes 3.Insertion (inordinacy) Search time= 0.000034300 sec Space = 4952 bytes	1.Dynamic Array Creation time = 0.005659400 sec Space=8248 bytes 2.Sorting time =0.000130600sec Space =4872bytes 3.Insertion (inordinacy) Search time= 0.000047200sec Space = 8248 bytes	1.Dynamic Array Creation time = 0.007797100sec Space=4936 2.Sorting time =0.000129500sec Space =4872bytes 3.Insertion (inordinacy) Search time= 0.000047000 sec Space = 4936 bytes	1.Dynamic Array Creation time = 0.000746500 Space= 35674 bytes 2.Sorting time = 0.000485400 sec Space = 35674 bytes 3.Insertion (inordinacy) Search time= 0.000021400 sec Space = 35725 bytes
Testcase4 =370104 words	1.Dynamic Array Creation time = 553.776635900 Space= 2960952 bytes 2.Sorting time = 0.049903000 sec Space = 2960888 bytes 3.Insertion (padawan) Search time= 0.000046100 sec Space = 2960952 bytes	1.Dynamic Array Creation time = 0.147903400 sec Space= 4194360 bytes 2.Sorting time = 0.046384500 sec Space = 2960888 bytes 3.Insertion (padawan) Search time= 0.000060900 sec Space = 4194360 bytes	1.Dynamic Array Creation time = 0.178160100 sec Space= 4113888 bytes 2.Sorting time = 0.045387500 sec Space = 2960888 bytes 3.Insertion (padawan) Search time= 0.000041200 sec Space = 4113888 bytes	1.Dynamic Array Creation time = 0.146842900 sec Space= 21961623 bytes 2.Sorting time = 0.149378400 sec Space = 21961623 bytes 3.Insertion (padawan) Search time= 0.000053400 sec Space = 21961671 bytes

From this chart I can say that incremental process takes lots of time to create dynamic array for larger data set, so in that case its complexity is $O(n^2)$, others method takes approximately similar time.

Phase 2.3: Risk Analysis

There are no risks associated with this application.

Phase 2.4: Verification

The program was tested multiple times by different inputs and gave the correct output result.

Phase 2.5: Coding

The code of this program is included in the zip file. The code is explained by in line comments

Phase 2.6:

I added an Input Output file to the zip file for several testcases.

Phase 2.7: Refining the program

No refinements are needed for this program.

Phase 2.8: Production

A zip file including the source files from eclipse and the output of the program have been included.

Phase 2.9: Maintenance

This application can be further improved once feedback from the grader is obtained.

3. Challenges

- **Efficient Handling of Large Datasets**: Dealing with large arrays and measuring performance for varying resizing strategies.
- **Fibonacci Resizing**: Computing Fibonacci numbers dynamically added overhead, making this strategy less efficient than expected.

4.Conclusion

The project successfully implemented a dynamic array with three different resizing strategies and empirically analyzed their time and space complexity. The **doubling strategy** was found to be the most efficient in terms of both time and space, providing a good balance between performance and memory overhead. The **incremental strategy** was inefficient, particularly for large datasets, and the **Fibonacci strategy** introduced unnecessary overhead due to Fibonacci number calculations. Python lists time and space complexity is like Doubling strategy.

Declaration:

This is my own project. I got help from the internet to finish it.

I give permission to the instructor to share my solution(s) with the class.