CS 261 Recitation 4: Analysis of Array Resizing and Binary Search

In order to get credit for the recitation, you need to be checked off by the end of recitation. For non-zero recitations, you can earn a maximum of 3 points for recitation work completed outside of recitation time, but you must finish this recitation before the next recitation. For extenuating circumstance, contact your recitation TAs and Instructor.

Group work, and individual work are highlighted

Recitation 4 Grade Breakdown:

Part 1: Dynamic arrays vs. linked lists for real-time systems	2 pts
Part 2: Dynamic Array Resizing Strategy	4 pts
Part 3: Find median of two sorted array	4 pts

Part 1: Dynamic arrays vs. linked lists for real-time systems

A real-time system is one in which every operation has an associated time deadline, and the system must complete each operation before its respective deadline, regardless of system load. Typically, these deadlines are tight, since real-time systems must continue to respond to changing conditions and inputs.

A robot is a good example of a real-time system. In particular, a robot continuously receives input from cameras, microphones, and other sensors, and it must quickly decide how to respond to each new piece of sensory input in order to act in the world in real time. If the robot takes too long to respond to a piece of sensory input, that input may become stale. Because conditions might be rapidly changing, an action or decision based on stale sensory input might prove disastrous for the robot. Thus, it is critical that the robot responds to each new piece of sensory input within a tight deadline.

Imagine you are a developer working on software to power a real-time system, like a robot, and you are currently working on a logging system to help debug problems with the larger real-time system. As the real-time system runs, your logging system will regularly generate new log entries, and all of the log entries must be stored as a collection within an appropriate data structure. The logging system must satisfy the following criteria:

- Periodically, your logging system will iterate through all of the stored log entries and write them all into a file, so other developers can use the logs to help debug the real-time system. The log entries will be written to the file in the same order they were collected.
- Because your logging system will only be responsible for writing log entries to a file in sequential order, it does not need to provide direct access to any specific log entry.

- Because you can't predict how long the real-time system will execute on any given run, you can't know exactly how many log entries will be generated. Longer runs will generate more log entries, and shorter runs will generate fewer entries. Still, you need to use memory efficiently, since the real-time system has limited memory available.

You are debating whether to use a dynamic array or a linked list to store the collection of log entries within your logging system.

(2 pts) Given the situation, which of these two data structures would you choose, and why? As a group, write your answer down. Make sure to clearly identify all relevant aspects of the problem that influenced your choice and to describe how your chosen data structure satisfies the constraints of the problem.

Part 2: Dynamic Array Resizing Strategy

Download and unzip the start code for this part:

https://classes.engr.oregonstate.edu/eecs/summer2022/cs261-001/recitations/rec4.zip

The code in the start code above contains a dynamic implementation, similar to the one you implemented in assignment 1. It contains these files:

- **`dynarray.h`** – This file contains the user-facing declarations for the dynamic array implementation. The user of the dynamic array will `#include` this file to use the dynamic array.
- **`dynarray.c`** – This file contains definitions for the structures and functions associated with the dynamic array.
- **`time_dynarray_insert.c`** – This file contains a simple program that times how long it takes to insert values into a dynamic array. We'll look more at exactly what this program does in a second.

(0.5 pt) Step 1: Compile and run the timing program

Compile and run the dynamic array code you just downloaded. You can use the included makefile to compile:

```
$ make
```

Now, run the timing program:

```
$ ./time_dynarray_insert
```

You should see the program generate a lot of output as it runs. The program specifically calculates how long it takes to insert some number of values (*n*) into a dynamic array. It does this for lots of different sizes of *n*, starting with *n* = 10,000 and going up to *n* = 1,000,000 in steps of 10,000. Each line of the program's output lists two different values, separated by a tab (`\t`):

```
<n>  <time>
```

Here, `<n>` specifically represents the value of *n*, and `<time>` indicates the time (in seconds) it took to perform *n* insertions into the dynamic array.

(1.5 pts) Step 2: Try some different array resizing strategies

Out of the box, the dynamic array implementation you're working with doubles the size of the data array each time it determines that the array needs to be resized. You can see this happening starting at line 107 in `dynarray.c`.

Modify the code to try **at least 3** different resizing strategies. For each testing strategy you implement, re-compile the code, and run the timing program. Note that some resizing strategies will result in the dynamic array's insert operation running *much* more slowly, and you might not have enough time in this recitation to run the timing program to completion for some particularly slow strategies.

(1 pts) Step 3: Compare resizing strategies

After you've implemented and tested some resizing strategies, compare the results of your testing. Specifically, look at the timing results for your various resizing strategies to see how they compare with each other.

Note that comparing the timing results of different resizing strategies might be hard to do by just looking at the raw numbers output by the timing program. To get a better picture of what those results look like and how they compare to each other, it might be useful to plot them.

The tab-separated output from the timing program is formatted to be easy to plot. In particular, you can save the output of the timing program into a TSV (tab-separated value) file, e.g. `results.tsv` using output redirection when you run the timing program or you just by copying/pasting the timing program's output. However you do it, be sure to save results for different resizing strategies to separate TSV files.

Once you have the timing data in a TSV file, it should be easy to import that data into an app like Google Sheets or Excel that can generate plots from data. There's also a basic plotting script included in the dynamic array folder you're working with, and you can use this to plot the timing results as a line graph.

The plotting script itself is called `generate_plot_from_tsv`. To use it to generate a JPEG image containing a plot of the data in a TSV file, first modify the following lines of the plotting script to specify the exact name of your input TSV file and the exact name of the JPEG file where you want to save the plot image:

```
INFILE="results.tsv"
OUTFILE="plot.jpg"
```

Once those lines are set correctly, you can run the plotting script from the command line like so:

```
$ ./generate_plot_from_tsv
```

After doing this, the line graph will be saved in a file called `plot.jpg`. You can repeat this process for different input and output files to generate plots for multiple datasets.

Importantly, note that the plotting script assumes that you have Gnuplot installed. It should already be installed on the ENGR servers, so you should be able to run the script there.

(1 pts) Step 4: Summarize your comparison
Finally, write a report summarizing your comparison. You can use some of the questions below as writing prompts for your summary, but feel free to write about anything you believe is relevant.

- Which resizing strategy resulted in the fastest runtimes?
- Which resizing strategy resulted in the slowest runtimes? Were there any that were too slow to collect meaningful data for?
- Why do you think various resizing strategies behaved the way they did?
- What kinds of tradeoffs are associated with the various resizing strategies? For example, do some strategies potentially use a lot more extra memory than others?
- Why do you think the most common resizing strategy is to double the capacity of the array on each resize?
- Based on the timing results for your different resizing strategies, what do you think is the average runtime complexity of the dynamic array's insert operation under each strategy?
    - Remember, each line of the timing program's output reports the total time to perform *n* insert operations, so to figure out the average runtime for an individual insert operation, you'll have to divide that total time by *n*.

If you have plot images depicting the timing results for different resizing strategies, feel free to include them in your summary if you think they help to illustrate the points you're making.

Part 3: Find the Median of Two Sorted Array

(2 pts) Problem Statement: Given two sorted arrays **`nums1`** and **`nums2`** of size **`nums1Size`** and **`nums2Size`** respectively, return the median of the two sorted arrays.

Example 1:

```
Input: nums1 = [1,3], nums2 = [2]

Output: 2.00000

Explanation: merged array = [1,2,3] and median is 2.
```

Example 2:

```
Input: nums1 = [1,2], nums2 = [3,4]

Output: 2.50000

Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.
```

Assuming;

- nums1.length == nums1Size
- nums2.length == nums2Size
- 0 <= nums1Size <= 1000
- 0 <= nums2Size <= 1000
- 1 <= nums1Size + nums2Size <= 2000
- -10^6 <= nums1[i], nums2[i] <= 10^6

(2 pts) Optimize your solution
Optimize your solution so that the overall run time complexity should be
O(log (nums1Size + nums2Size)).

Make sure you get checked off by showing them the output of your program, your report, and your group work before the end of your recitation section.

For backup purposes, please submit your work for this recitation (including all documents/text files for group work, and programs) to TEACH.

If you finish your recitation early, it is recommended that you stay and work on your assignment 2 to get feedback.