Week 5 Recitation Exercise

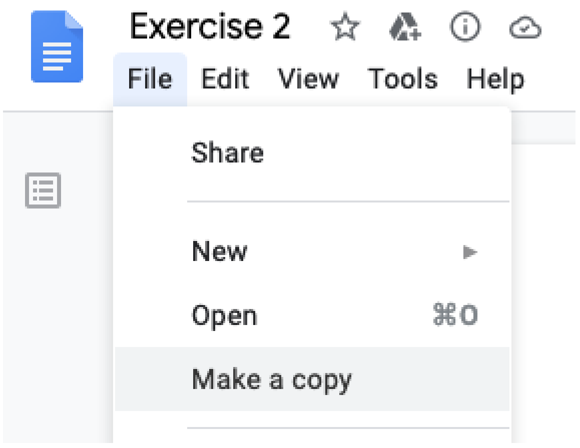
CS 261 – 10 points

In this exercise, you will explore the computational complexity of the dynamic array's insert operation. You'll specifically examine how the dynamic array implementation's resizing strategy impacts the overall performance of the insertion operation.

Follow the instructions below, and when you're finished, make sure to submit your completed exercise on Canvas. Please submit one copy per group (as long as all group members' names are listed below, it's fine if just one person submits). Remember that this exercise will be graded based on effort, not correctness, so don't worry if you don't get all the right answers. Do make sure you *try* to get the right answers though.

Step #1: Organize your group and create your submission document

Your TA will assign you to a small group of students. Among your group, select one group member to record your group's answers for this exercise. This group member should make a personal copy of this document so they are able to edit it:



Step #2: Add your names

Next, your group's recorder should add the names and OSU email addresses of all of your group's members in the table below:

Group member name	**OSU email address**

Hyun Taek, Oh	ohhyun@oregonstate.edu
Hua Hsu	hsuhua@oregonstate.edu
Qianping He	heqia@oregonstate.edu

Step #3: Grab some dynamic array code

Now, take a look at the code in the following GitHub repo:

https://github.com/osu-cs261-f23/recitation-5

Clone this repo onto your own machine or onto one of the ENGR servers, but note that you won't be able to push changes back to the repo, since your access to it is read-only. If you'd like to be able to push any changes you make to the code back to GitHub, you'll have to make a fork first and then clone and work with the fork.

The code in the repo above contains a dynamic array 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.

Step #4: Compile and run the timing program

Compile and run the dynamic array code you just cloned to your machine. 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.

Step #5: Try some different dynamic 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 a few different resizing strategies. For example, you could add some constant value *k* to the array capacity each time the array is resized (i.e. `capacity += k`) or you could multiply the capacity by a factor *k* different from 2 (i.e. `capacity *= k`). For both of these different strategies, you can also experiment with different values of *k*. You might be able to come up with other strategies as well. Feel free to be creative.

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.

Step #6: 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 repo you're working with, and you can use this to plot the timing results as a line graph if you want to.

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. Gnuplot should already be installed on the ENGR servers, so you should be able to run the script there.

Step #7: Summarize your comparison

Finally, write a few paragraphs in the space below 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?

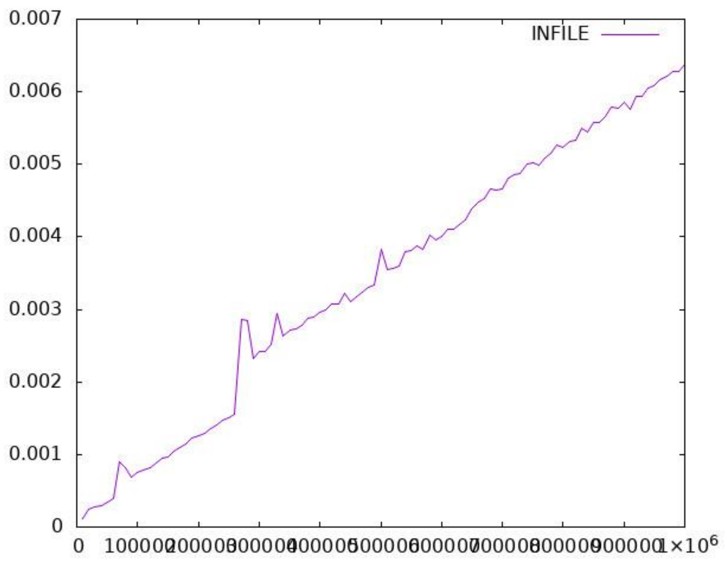
- What kinds of tradeoffs are associated with the various resizing strategies? For example, do some strategies potentially use a lot more extra memory than others?
- Why do you think the most common resizing strategy is to double the capacity of the array on each resize?
- Based on the timing results 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 below if you think they help to illustrate the points you're making.

Summary:

[1. Which resizing strategy resulted in the fastest runtimes?

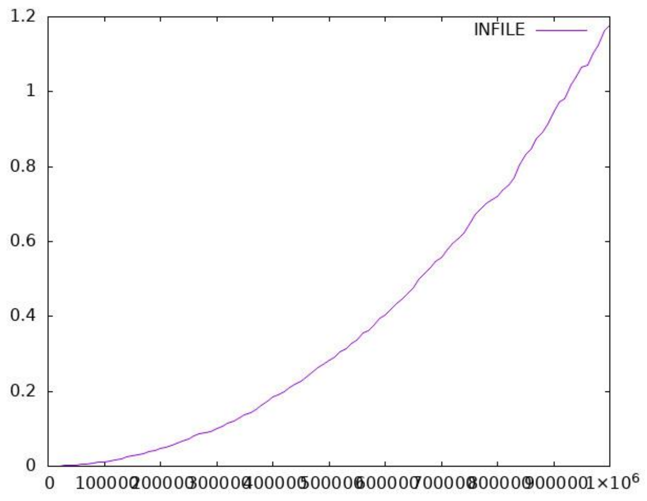
: Capacity *= k (The higher the value of k is, the faster and less time is taken)



The case of "k=4"

2. Which resizing strategy resulted in the slowest runtimes? Were there any that were too slow to collect meaningful data for?

: Capacity += k



The case of “k=1000”

: Unlike * k method, especially, if the value of k is too low and the increase in capacity by adding space, it takes too much time and even not end
(you can see the total time differences between * method and + method
e.g. * method: 0.04ms // + method: 1.2ms)

3. Why do you think various resizing strategies behaved the way they did?

: When data insertion happen, the program checks the empty space of dynamic array and resize it. If the times of resizing happen a lot, the total time cost will increase, so it takes so long time.

4. 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?

: The best case is to know how much space the program needs. However, we do not know exact size. + k method is kind of little steps to make space, which means many times to call resizing function. * k method is kind of big steps, which means a program may cause a waste of memory space. So, potentially * k method use a lot more extra memory than others, especially the value of k is large enough.

5. Why do you think the most common resizing strategy is to double the capacity of the array on each resize?

: It is needless to say that + k method is not good for resizing strategy, causing the program to call resizing function a lot if input data is large enough. The differences between k=2 and 3 are quite obvious, of course, k=3 method takes less time than k=2 method but there are so many memory waste. It seems trade-off between the degree of resizing and memory waste. Thus, doubling capacity can be the most common resizing strategy.

6. 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?

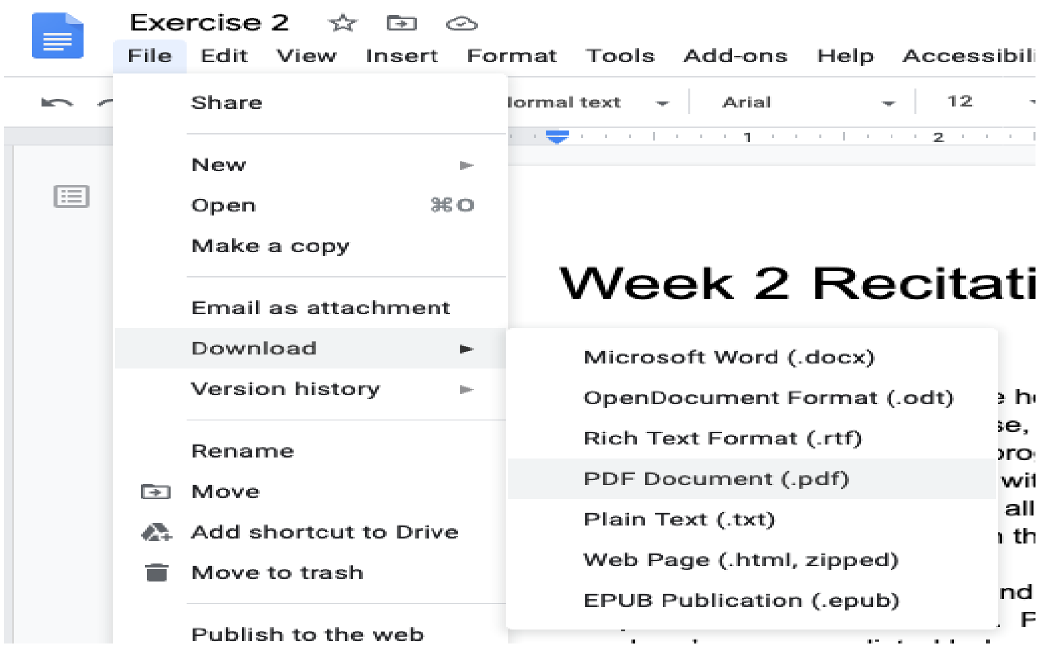
* Remember, each line of the timing program's output reports the total time to perform n insert operations, so to figure out the average runtime for an individual insert operation, you'll have to divide that total time by n.

: Dynamic array spend time to write value into an array and copy(resize) the capacity of the array when there is no space in the array. In other words, overall cost of writing the value (O(n)) and copying(resizing) the capacity of array (O(n-1)) is O(2n-1). Then, the amortized cost and average runtime complexity of dynamic array insertion is (2n-1) / n = O(1). Thus, on average, data insertion in the case of dynamic array is a constant time operation.

]

Submission and wrap-up

Once you've completed this exercise, download your completed version of this worksheet as a PDF and submit that PDF on Canvas (refer to the image below to help find how to download a PDF from Google Docs). Again, please submit one copy of your completed worksheet for your entire group. It's OK if just one member submits on Canvas. All group members will receive the same grade as long as their names and email addresses are included above.


Exercise 2
File
Edit
View
Insert
Format
Tools
Add-ons
Help
Accessibili
lormal text
Arial
12
Share
New
Open ⌘O
Make a copy
Email as attachment
Download
Version history
Rename
Move
Add shortcut to Drive
Move to trash
Publish to the web
Microsoft Word (.docx)
OpenDocument Format (.odt)
Rich Text Format (.rtf)
PDF Document (.pdf)
Plain Text (.txt)
Web Page (.html, zipped)
EPUB Publication (.epub)
Week 2 Recitati