Project 1: implementing algorithms (40 points)

CPSC 335 - Algorithm Engineering
Fall 2020
Instructor: Doina Bein (dbein@fullerton.edu), Mike Peralta (miperalta@fullerton.edu)

Abstract

In this project you will set up an environment for implementing algorithms in C++, and use that to implement two algorithms that solve the same problem. The first step is for you to create your own Tuffix Linux install. We will use this environment to submit and grade projects all semester. The second step is for you to download the starter code from GitHub, complete the code for each algorithm, translate descriptions of two algorithms into pseudocode; analyze your pseudocode mathematically; implement each algorithm in C++; test your implementation; and describe your results. The third step is to do research on the Internet about the usefulness of having a .gitignore file and create the file .gitignore in your directory. Then upload everything back on GitHub.

Installing Tuffix

You can set up Tuffix as a native install on a dedicated computer, or as a virtual machine (VM), by following the Tuffix Installation Instructions. You can use your own computer, or borrow a computer from CSUF for free through the Long-Term Laptop Checkout process.

Another option is CSUF's Virtual Computing Lab (VCL). This allows you to connect remotely to a Tuffix-like environment. Keep in mind a VCL session only lasts a limited time (up to 4 hours) and *is erased at the end of the session,* so you *must save your work elsewhere before your session ends* (e.g. `git push` all your work before the session ends).

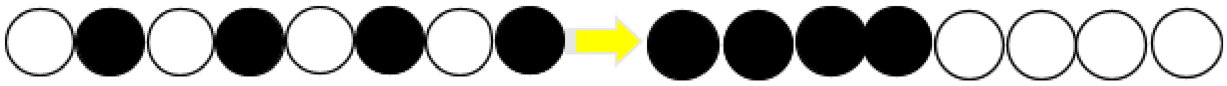
Students using Tuffix should join the CSUF TUFFIX slack workspace at https://csuf-tuffix.slack.com. Please use the `#general` channel to ask about troubleshooting, installing, and using Tuffix.

The smoothest and best-performing option is to install Tuffix natively on a computer, so that is our first recommendation. If that is unfeasible for you, use a VM or VCL. The VM option requires a more powerful computer (especially RAM memory) but saves work indefinitely. VCL requires a stable internet connection and does not save work between sessions.

The Alternating Disk Problem

The problem, changed from the one presented in Levitin's textbook as Exercise 14 on page 103, is as follows:

> You have a row of $2n$ disks of two colors, n light and n dark. They alternate: light, dark, light, dark, and so on. You want to get all the dark disks to the left hand end and all the light disks to the right hand end. The only moves you are allowed to make are those that interchange the positions of two neighboring disks. Design an algorithm for solving this puzzle and determine the number of moves it takes.



The alternating disks problem is:

Input: a positive integer n and a list of $2n$ disks of alternating colors light-dark, starting with light
Output: a list of $2n$ disks, the first n disks are dark, the next n disks are light, and an integer m representing the number of swaps to move the light ones after the dark ones

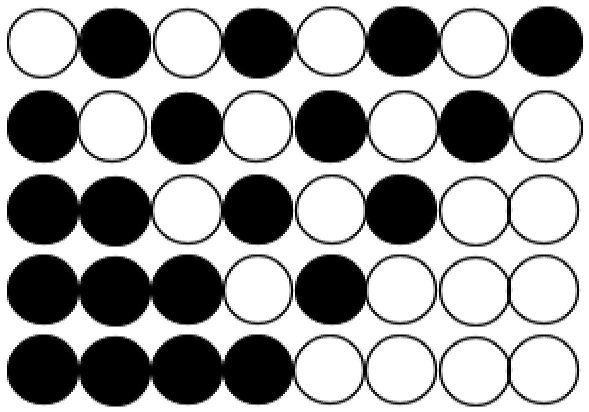
There are two algorithms, presented below, that solve this problem in in $O(n^2)$time. You need to translate these descriptions into clear pseudocode.

The first algorithm proceeds like a lawnmower: starts with the leftmost disk and proceeds to the right until it reaches the rightmost disk: compares every two adjacent disks and swaps them only if necessary. Now we have one darker disk at the left-hand end and one lighter disk at the right-hand end. Once it reaches the right-hand end, it starts with the rightmost disk, compares every two adjacent disks and proceeds to the left until it reaches the leftmost disk, doing the swaps only if necessary. The lawnmower movement is repeated $\lceil n/2 \rceil$ times. Some improvement can be obtained by not going all the way to the left or to the right, since some disks at the ends are already in the correct position.

The second algorithm has n runs. In each run, every two disks are compared and swap only if necessary. The first run compares the first and second disk, the third and fourth disk, the fifth and the sixth disk, etc.. The second run compares the second and third disk, the fourth and the fifth, the sixth and the seventh disk, etc.. The third run is a repetition of the first run. The fourth run is a repetition of the second run, etc.. Again, swaps are done only if necessary. Run 1, 3, etc. starts with the leftmost disk and proceeds to the right until it reaches the rightmost disk: compares every two adjacent disks and swaps them only if necessary. Now we have one darker disk at the left-hand end and the lighter disk at the right-hand end. Run 2, 4, etc. starts with the second leftmost disk and proceeds to the right until it reaches the second rightmost disk: compares every two adjacent disks and swaps them only if necessary. There are a total of n runs.

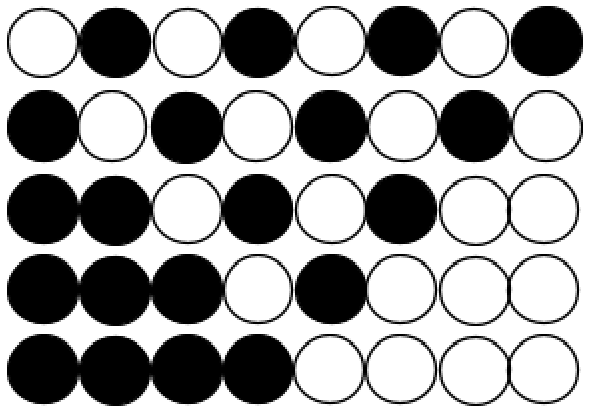
The lawnmower algorithm

It starts with the leftmost disk and proceeds to the right, doing the swaps as necessary. Now we have one lighter disk at the left-hand end and the darker disk at the right-hand end. Once it reaches the right-hand end, it starts with the disk before the rightmost disk and proceeds to the left, doing the swaps as necessary, until it reaches the disk before the left-hand end.
The lawnmower movement is repeated ⌜n/2⌝ times.
Consider the example below when n=4, and the first row is the input configuration, the second row is the end of comparison from left to right, the third row is the end of the first run (round trip that contains left to right followed by right to left), etc.. The exact list of disks changes as follows at the end of eachrun (we consider a run to be a check of adjacent disks from left-to-right or right-to-left) is shown below:



The alternate algorithm

It starts with the leftmost disk and proceeds to the right until it reaches the rightmost disk: compares every two adjacent disks and swaps them only if necessary. It does not iterate through each index, but iterates over each ***pair*** (i.e., it moves 2 steps at a time). Now we have one darker disk at the left-hand end and the lighter disk at the right-hand end. This is the end of Run 1.
Next it starts with the second leftmost disk and proceeds to the right until it reaches the second rightmost disk: compares every two adjacent disks and swaps them only if necessary. This is the end of Run 2.
Next it is Run 3, that proceeds exactly as Run 1, starting with the leftmost disk. Run 3 is followed by Run 4 that is exactly as Run 2, starting with the second leftmost disk. So really, Run 1 and Run 2 continually alternate until sorting has finished.
There are a total of n runs.
Consider the example below when n=4, and the first row is the input configuration, the second row is the end of comparison from left to right, the third row is the end of the first run (round trip that contains left to right followed by right to left), etc.. The exact list of disks changes as follows at the end of eachrun:



Algorithm Design

Your first task is to design an algorithm for each of the two problems. Write a clear pseudocode for each algorithm. This is not intended to be difficult; the algorithms I have in mind are all relatively simple, involving only familiar string operations and loops (nested loops in the case of oreos and substrings). Do not worry about making these algorithms exceptionally fast; the purpose of this experiment is to see whether observed timings correspond to big-O trends, not to design impressive algorithms.

Mathematical Analysis

Your next task is to analyze each of your two algorithms mathematically. You should prove a specific big-O efficiency class for each algorithm. These analyses should be routine, similar to the ones we have done in class and in the textbook. I expect each algorithm's efficiency class will be one of $O(n)$, $O(n^2)$, $O(n^3)$, or $O(n^4)$.

Obtaining and Submitting Code

This document explains how to obtain and submit your work:

GitHub Education / Tuffix Instructions

Here is the invitation link for this project:
https://classroom.github.com/g/BTVsuhUr

Implementation

You are provided with the following files.

1. `disks.hpp` is a C++ header that defines functions for the two algorithms described above. There are also classes that represent the input and output of the alternating disk problem. The function definitions are incomplete skeletons; you will need to rewrite them to actually work properly.
2. `disks_test.cpp` is a C++ program with a `main()` function that performs unit tests on the functions defined in `disks.hpp` to see whether they work, prints out the outcome, and calculates a score for the code. You can run this program to see whether your algorithm implementations are working correctly.
3. `rubrictest.hpp` is the unit test library used for the test program; you can ignore this file.
4. `README.md` contains a brief description of the project,and a place to write the names and CSUF email addresses of the group members. You need to modify this file to identify your group members.

What to Do

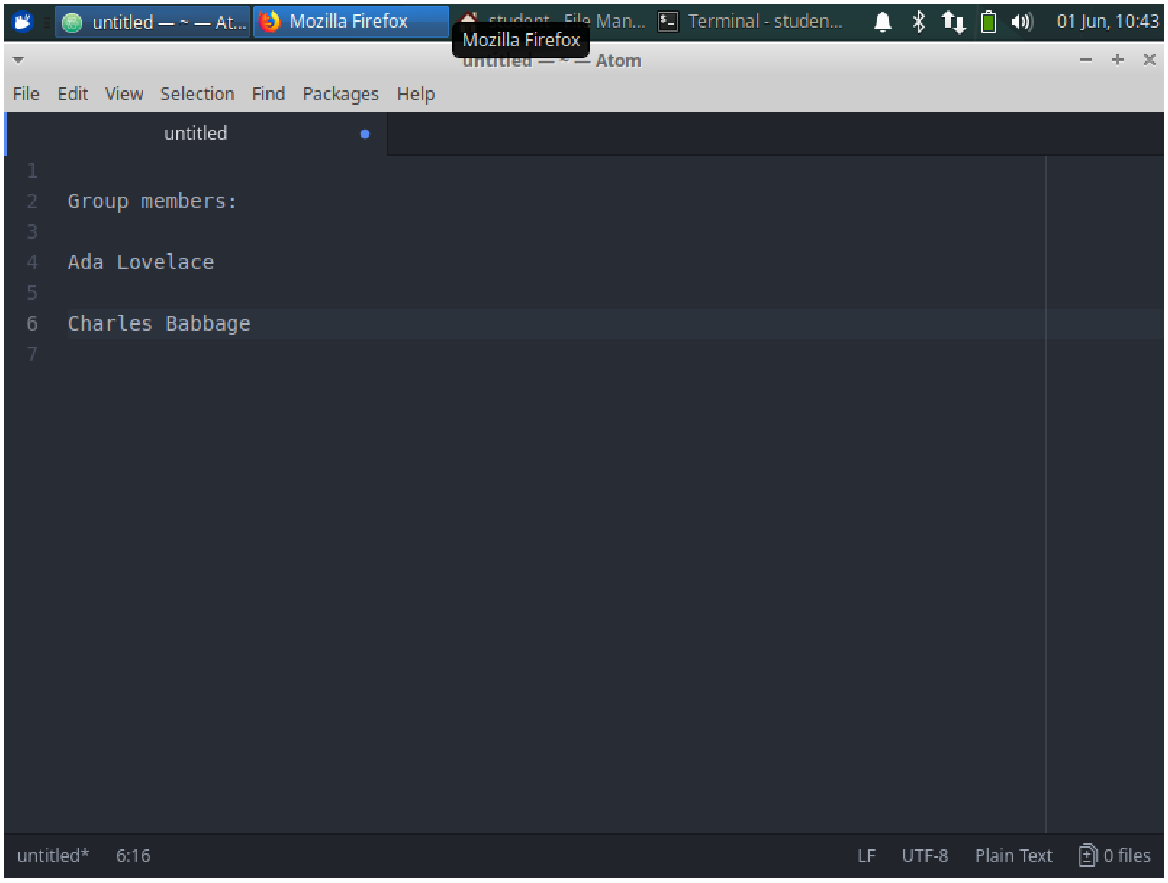
First, add your group member names to `README.md`. Then:

1. Write your own pseudocode for the lawnmower algorithm, and the alternating algorithm.
2. Analyze your pseudocode for the algorithm mathematically and prove its efficiency class.
3. Implement all the skeleton functions in `disks.hpp`. Use the `disks_test.cpp` program to test whether your code works.
4. Create the file .gitignore for this project in order not to track the binary (executable) file(s) produced by executing "make". In this case, the binary file produced will be "disks_test".

Finally, produce a brief written project report ***in PDF format***. Submit your PDF by committing it to your GitHub repository along with your code. Your report should include the following:

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 1.
2. Two full-screen screenshots: one inside Tuffix, showing the Atom editor, with your group member names inside Atom. One way to make your names appear in Atom is to simply open your `README.md`. The second screenshot is with your code executing the command `make`.
3. Two pseudocode listings, for the two algorithms.
4. A brief proof argument for the time complexity ofyour two algorithms.

Your screenshot might look like this:



```
Group members:

Ada Lovelace

Charles Babbage
```

Grading Rubric

Your grade will consist of three parts: *Form, Function,* and *Analysis.*

Function refers to whether your code works properly as defined by the test program. We will use the score reported by the test program, when run inside the Tuffix environment, as your Function grade.

Form refers to the design, organization, and presentation of your code. A grader will read your code and evaluate these aspects of your submission.

Analysis refers to the correctness of your mathematical and empirical analyses, scatter plots, question answers, and the presentation of your report document.

The grading rubric is below.

1. Function = 6 points, scored by the unit test program
2. Form = 11 points, divided as follows:
    a. README.md completed clearly = 3 points
    b. Style (whitespace, variable names, comments, helper functions, etc.) = 3 points

    c. C++ Craftsmanship (appropriate handling of encapsulation, memory management, avoids gross inefficiency and taboo coding practices, etc.) = 3 points
    d. File `.gitignore` created and populate it correctly **= 2 points**

3. Analysis = 23 points, divided as follows
    a. Report document presentation = 3 points
    b. Screenshots = 3 points each (total 6 points)
    c. Pseudocode = 3 points each (total 6 points)
    d. Mathematical analysis for each pseudocode = 4 points each (total ~~10~~ **8** points)

Legibility standard: As stated on the syllabus, submissions that cannot compile in the Tuffix environment are considered unacceptable and will be assigned an "F" (50%) grade.

Deadline

The project deadline is Friday, September 25, 11:59 pm on GitHub.

You will be graded based on what you have pushed to GitHub as of the deadline. Commits made after the deadline will not be considered. Late submissions will not be accepted.