Assignment Analysis of Algorithms

COSC 2336: Data Structures and Algorithms

Spring 2024

Objectives

- Program examples of linear and exponential algorithms.
- Experience practical examples of the difference in performance that algorithms can have on runtimes.
  - In particular, see differences of $\mathcal{O}(1)$, $\mathcal{O}(n)$ and $\mathcal{O}(2^n)$ constant time, linear and exponential performance.
- More examples of and practice with writing recursive functions.
  - Here though we will see an efficient exponential recursive algorithm..
- Better understand algorithm complexity and the impact algorithms have on computational performance.

Description

In this assignment we will explore some more aspects of algorithmic complexity and the practical analysis of code that implements an algorithm to perform some task or function. This week you will be implementing several algorithms to compute the Fibonacci Sequence. The n^{th} Fibonacci number is computed as the sum of the previous two Fibonacci numbers, given initial conditions:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned} \tag{1}$$

This defines the following sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 \tag{2}$$

For example, the 0^{th} Fibonacci number is 0, the 1^{st} Fibonacci number is 1 and the 10^{th} Fibonacci number is 55. In this assignment you will be writing several regular C functions to compute and return the n^{th} Fibonacci number of the Fibonacci sequence. You will implement constant time, linear time and exponential time algorithms of this calculation, and will also use a technique known as memoization to perform the Fibonacci calculation. You will then investigate and compare the performance of these different algorithmic implementations.

Note: We will be using regular signed integers to hold and calculate the Fibonacci numbers. Fibonacci numbers are an example of a type of exponential growth, the size of the Fibonacci numbers in the sequence grows rapidly. In fact, a regular signed integer on most common environments uses 32 bits to represent numbers, where 1 of the bits is used as a sign bit, leaving 31 bits to represent the magnitude of the number. This gives a range of approximately $\pm 2^{31} = \pm 2,147,483,648$ or a bit over 2 billion. In fact $F_{46} = 1,836,311,903$ which is the largest number we can hold in a standard signed int, as $F_{47} = 2,971,215,073$ which overflows the 31 bit magnitude we have for representing integers. So we will be restricting ourselves to generating and testing Fibonacci numbers up to the 46^{th} number of the Fibonacci sequence.

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/assg06-tests.cpp`	Unit tests and performance tests of the Fibonacci algorithm implementations
`include/libfibonacci.hpp`	Header file for function declarations of the Fibonacci algorithms
`src/libfibonacci.cpp`	Implementation file for the algorithm implementations of the various Fibonacci calculations

This week you will be implementing several regular C functions to compute and return the n^{th} value of the Fibonacci sequence. As in previous assignments, you will be putting all of your function declarations into the `libfibonacci.hpp` header file, and all implementations of the functions should reside in the `libfibonacci.cpp` source file.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment Recursion' for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment. Make sure to open the cloned folder and restart inside of the correct Dev Container.
3. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also make sure you link the issues with the `Feedback` pull request.

Assignment Tasks

You will be creating 4 different versions of the computation of the Fibonacci sequence. You will create a linear time algorithm, a constant time algorithm and an exponential time algorithm. In addition we will use the memoization technique to create a look up table of Fibonacci numbers and look at its performance.

In addition to creating the 4 different algorithmic implementations, you will also measure the performance of the functions. You will look at the actual wall-clock-time each takes to perform its calculations, as well as counting operations of your implementations as one does to compare algorithmic complexity.

Task 1: Implement Linear Fibonacci Algorithm

In some ways the linear algorithm for calculating the n^{th} Fibonacci number may be the most complicated of the 4 approaches you will implement with this task. But we will start with this one as it is still probably the most familiar approach, though at this point you have practiced recursion enough that you may find the recursive version in Task 2 actually easier to comprehend and implement now.

For a linear algorithmic approach, you need to write a loop that can keep track of the previous two Fibonacci numbers, so that you can sum them up and calculate the next one. Then you need to keep doing this by shifting the $n-1$ number to become the $n-2$ number, and the n number you just calculated to be the $n-1$ number, so that the next iteration of the loop you can calculate the next number in the sequence.

So for this task, create a function named `fibonacciLinear()`. This function takes a single integer value `n` as input, which indicates which Fibonacci number in the sequence is to be calculated and returned. This function returns an `int` result. In fact all of the 4 functions you will write for this assignment have identical signatures, they take an `int n` as input, and return the n^{th} Fibonacci number as an `int` result from the function.

The best approach for both the linear and recursive implementations has some similarity. For the linear (and recursive) algorithm, start by handling the special initial conditions. When $n = 0$ return `0` and when $n = 1$ return `1`. Or alternatively when $n <= 1$ return `n`, though these two different tests have different results for error conditions. However in this assignment we will not test for nor define what happens if a $n < 0$ is given to one of our Fibonacci algorithms.

But if we are not calculating one of the initial conditions, then we have been asked to calculate some Fibonacci number for $n >= 2$. You should do this by implementing a loop using the following algorithm/approach.

1. Initialize 3 local variables to hold the $n-2$, $n-1$ and n Fibonacci numbers. We will be starting by calculating F_2, so you should initialize the $n-2$ value to be $F_0 = 0$ and the $n-1$ value to be $F_1 = 1$.
2. Implement a loop that iterates from index $i = 2$ up to $i = n$, where n was the input given of which Fibonacci number to calculate and return. At this point since you already handled $n = 0$ and $n = 1$, then n should be $n >= 2$ at this point.
   - Calculate the current n^{th} Fibonacci number using the values in $n-1$ and $n-2$ variables.
   - Copy the $n-1$ to the $n-2$ value and the n you just calculated to the $n-1$. This shifting is in preparation for the next loop iteration, if more loop iterations are needed.
3. Return the final calculated n Fibonacci number from the loop.

If you defined your loop correctly and shift your local variables as described, the loop will run the correct number of times needed to calculate the n^{th} Fibonacci number of the sequence. For example when $n = 2$, the loop will execute only a single time, and will calculate n from the initial values of $n-1$ and $n-2$ you gave to the local variables. If $n = 3$ then the loop executes 2 times, and you will calculate and shift down the values in the first iteration so that on the second iteration you can calculate F_3.

Once your linear implementation is working and passing the tests, commit your changes and push them to your class GitHub repository with an appropriate commit message.

Task 2: Implement Recursive Fibonacci Algorithm

Make sure you have created Issue 2 for this task and linked it with the `Feedback` pull request before beginning.

The recursive version of the Fibonacci calculation is in many ways simpler to implement than the linear version with a loop. The function will be named `fibonacciRecursive()` for this task, but as mentioned before the signature of the function will be the same as for the linear algorithm, it takes an `int n` as input and returns an `int` result.

The base cases will be the same as the initial conditions you implemented for Task 1. When $n = 0$ return $F_0 = 0$ and when $n = 1$ return $F_1 = 1$, which are the 0^{th} and 1^{st} Fibonacci numbers of the sequence.

For the general recursive case, simply call `fibonacciRecursive()` recursively on $n-1$ and $n-2$, sum up these returned values and return that as the result of your recursive implementation.

As we will discuss in class, this recursive implementation requires approximately exponential time to perform its work, because it is very inefficient compared to the linear implementation. In fact, if you write out the recursion tree of this function, you will see that many values are calculated many times repeatedly of the Fibonacci numbers just to get a final result. In fact the recursion requires about φ^n function calls to be performed, where φ (Greek phi symbol) is the golden ratio (read the links above for more information). In fact this can take so much time that you will find your tests to calculate Fibonacci numbers of $n >= 40$ will be noticeably slow, and they have been commented out by default to start with. But you should uncomment them and try them out at least one time once you have your recursive function working, to see how long it takes to calculate for those values.

Once you have implemented your recursive Fibonacci calculation and it is passing the given tests, do the usual. Commit your changes and push them to the `Feedback` pull request of your GitHub repository.

Task 3: Empirical Performance of Linear and Recursive Implementations

Make sure you have done the usual to commit and document the previous task in your repository and you have created the issue for Task 3.

Tasks 3 and 6 are a little different than in previous assignments. There are two unit tests for task 3. For the first one, you don't have to do anything. In the first unit test of task 3 we demonstrate performing an empirical measurement of the wall-clock-time used to calculate the 40^{th} Fibonacci number using both the linear and recursive implementations. The recursive implementation is much less efficient than the linear implementation, needing exponential time to perform the calculation. You will see that each algorithm is run 5 times, and the average elapsed time is calculated, which is typical when performing empirical measurements of elapsed time. Taking an average of a number of runs will tend to lessen the impacts of extraneous system events that might have undue large impacts on a single run of the algorithm. You should note that the linear algorithm runs in nanoseconds, and the time it takes will be imperceptible to a person, but the recursive algorithm will take 100's of millions of nanoseconds, where a nanosecond is a billionth (10^{-9}) of a second. So typically on my system it takes close to a second to run 1 calculation of F_{40}, though your time will vary depending on your system capabilities.

The second unit test for task 3 does require some small work. There is a global variable named `operationCount` defined in the `test-algorithms.cpp` file that you may have noticed. In order to count up the number of operations or executions performed, we are going to add some code to sum up operations into your 2 Fibonacci functions. You will notice that in the second task 3 test it first initializes `operationCount` to 0 before calling your function, then reports the calculated `operationCount` after your function runs. This is typical of profiling libraries, that will add in similar code to be able to count up the number of entries and executions of each function and each line of code in your code base. And it is the same as counting up the number of operations performed to estimate algorithmic complexity, as has been discussed this week.

For your `fibonacciLinear()` function, calculate the number of operations performed by your code. You should try and count up all assignment statements, all arithmetic statements, and all boolean comparisons. There will be some number of operations that happen outside of the loop, which corresponds to the `c1` constant defined in this test. Also, if you implement your loop as described, you loop will execute $n - 1$ iterations, and `c2` corresponds to the number of operations that happen in the loop. Don't forget to count operations such as the loop comparison made to determine if the loop is done or not, and the increment you do for your loop index variable, which all happen 1 time for each loop iteration.

Your values for `c1` and `c2` might be slightly different than the ones given in the `test-algorithms.cpp` test file, depending on how you implemented your code. Usually you should not change the test code, only uncomment it. But here, if you need to, you can modify the `c1` and `c2` values to match your implementation and get the test to pass. But you should have similar values and a similar operation count for your linear implementation as shown in this test code.

For the recursive algorithm, to simplify things a bit and make the counts more understandable, lets simply attempt to count the number of times that the `fibonacciRecursive()` function is called when performing a computation. So you can easily add in this count by simply incrementing `operationCount` by ‘ each time your function is called.

If you implemented your recursive function as described, you should get exactly the same number of calls of your function as is expected in the test. The total number of times the recursive function will be called is related to φ^n as described, because Fibonacci numbers are deeply related to the golden ratio φ. The expression for the `expectedOperationCount` is the total number of all recursive calls that should be made by your implementation. And again you should note that fundamentally this number of recursive calls is an exponential function, similar to 2^n, so the amount of work grows exponentially in n, making this performance very slow for even relatively modest values of n we might want to calculate.

Once you are satisfied with your empirical performance tests, commit your work and push it to the `Feedback` pull request of your repository.

Task 4: Implement Constant Time Fibonacci Calculation

Perform the usual prerequisite steps before starting task 4.

For task 4 you will create a constant time implementation of the Fibonacci computation. As before, you will create a function named `fibonacciConstant()` for this task that takes the same input parameter and returns the same result as the previous two versions.

If you read down a bit in the Fibonacci Numbers article about the closed-form expression of the Fibonacci computation, Binet's formula is discussed. This gives an exact equation, or a closed-form expression, to calculate the n^{th} Fibonacci number. Since the calculation does not need to loop or recurse depending on the size of n, it will perform in constant time $\mathcal{O}(1)$ no matter what value of n you wish to compute.

Binet's formula closed-form expression is as follows:

$$F_n = \left[\frac{\varphi^n}{\sqrt{5}} \right], n \geq 0 \tag{3}$$

Note: here the symbol [] is being used as special notation to mean to round the value to the nearest whole integer. The fraction will result in a real valued number, that should be rounded to the closest integer to get the exact Fibonacci number result desired.

The golden ratio φ is not one of the constants defined in the `<cmath>` library, so we have added and defined it as a constant to use for this task at the top of `libfibonacci.cpp`. Also you will need to use the `round()` function from the `<cmath>` library to round your result to the closest integer value.

Depending on how you count operations, this expression only takes 5 or 6 operations to perform, if you count division, raising to a power, square root, and assignment, for example. But these 5 operations will always be the only ones needed to calculate any value of n of the Fibonacci sequence.

Once you are satisfied with your implementation of the constant time closed-form Fibonacci calculation and it is passing the tests, perform the usual steps to commit and push your work document to the `Feedback` pull request.

Task 5: Implement Fibonacci calculation using memoization

In the final task where you have to implement a new function, we will introduce you to a technique known as memoization, also some times simply called tabling. This is kind of a fancy name for what is conceptually a rather simple technique. The basic idea is that we want to have a look up table that we just have calculated, that contains the n^{th} Fibonacci result for each value of n that we need. If the values of the table have all already been calculated, the time complexity to look up the value is constant $\mathcal{O}(1)$, and would even be smaller than your constant time function, since a single lookup would be less expensive than the 5 or so arithmetic operations needed for that implementation.

In this task you will be required to use memoization combined with recursion, but the memoization table will effectively reduce the computational time to linear or better if we take into account repeated use of the calculated table. Memoization usually works by computing lookup table values as needed, so initially the table is emptied, and only when a value is requested do we take the time to perform the calculation, store it for any future use, and return it (just-in-time calculation).

In `libfibonacci.cpp` we have already defined a global `fibonacciTable`, which is a table of integer results, as well as a function named `initializeFibonacciTable()` that initializes the table. The table is initialized so that the initial conditions of $F_0 = 0$ and $F_1 = 1$ are precomputed and available in the table, but we use the defined symbol `NOT_CALCULATED` for all other table entries as an indication that this result has not been calculated yet.

Implement your `fibonacciMemoization()` function using the following approach. As with the other versions, this function task `n` an `int` as input, and calculates and returns and `int` result which is the n^{th} calculated Fibonacci number. The memoization implementation is easiest to perform using the following algorithm/steps:

1. First check the `fibonacciTable` to see if the requested value of `n` has not yet been calculated. If the value has not been calculated, perform the calculation.
   - Basically use a recursive approach, but call `fibonacciMemoization()` recursively on $n-1$ and $n-2$ to calculate/lookup the previous 2 Fibonacci numbers.
   - Sum up these and update the `fibonacciTable` with the result of this calculation when performed.
2. After check and calculating if needed, you are assured that the value of the n^{th} Fibonacci number is in the table, having either just been calculated, or calculated some time previously. So simply look up the value in the table and return it as the result.

Implement this method and get it to pass the unit tests for the Fibonacci memoization after uncommenting them in the test file. When you commit and push this work to the `Feedback` pull request, check that you are successfully passing all of the GitHub autograding actions as this is the last task for this assignment.

Task 6: Empirical Performance of Constant and Memoization Implementations

As before, there are two final unit tests to do some performance measures and empirical testing of the constant time and memoization versions of the Fibonacci computation. The first test you have nothing to do, but note the calculated empirical average elapsed time for the two new version, and compare with the timing of the previous two versions of the calculation. You should find that the constant time algorithm of course gives the best performance. But you may be a bit surprised to see that the memoization version performs almost as fast, and usually beats the linear time algorithm by 50% or so, which if you think about what is being done with the memoization recursive calls might be a bit surprising of a result, but here the overhead of the recursion is less than the overhead of the iterations of the loop being done. And this timing result is a bit unfair to memoization, as after the table is filled up, it will perform essentially in constant time, and be even faster than the constant time function that has to perform several operations to calculate the number each time.

But you do need to make some small changes for the second test of task 6 to empirically count up operations. Though the constant time algorithm is trivial in this case as you should simply count up and return the number of additions, raising to a power, divisions, function calls and assignments you do and set the `operationCount` to that value in your `fibonacciConstant()` function.

For the `fibonacciMemoization()` function, again there is a defined `c1` and `c2` values, that may be slightly different for you depending on how you implement your code, and you can tweak these a bit to match your implementation. But basically `c1` is intended to be the operation count for times when your function is called and the value needs to be calculated. This includes doing the calculation, assigning it to the table, then looking it up and returning it. `c2` is the count of operations when you function is called and the calculation does not need to performed, only looked up in the table. `c2` is probably 2 for most people, because you need to count the boolean test to see if you need to perform calculation as 1 operation, then the lookup from the table as another operation.

When you are satisfied with your code and are passing the tests for task 6, create and push a commit to your repository to complete the assignment.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 10 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 15 points are awarded for completing each of the 6 tasks. However you should note that the autograder awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
    - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
    - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).

4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Lecture U05-1 Searching
- Lecture U05-2 Sorting
- Lecture U06-1 Analysis of Algorithms
- Binary Search Playlist MyCodeSchool
- Sorting Algorithms Playlist MyCodeSchool
- Time Complexity Analysis Playlist MyCodeSchool
- Recursion Playlist MyCodeSchool
- Recursive Sorting Algorithms