LAB 2 REPORT

Comparing Time Complexity of Iterative and Recursive Implementations of Pascal's Triangle"

Author: Jakob Balkovec Seattle, WA

Instructor: Beatriz Diaz Acosta, MS Computer Science 04/05/2023

Table of Contents

[INTRO 3](#_Toc131620030)

[FUNCTIONS in C++ std=11 4](#_Toc131620031)

[Iterative Bico function 4](#_Toc131620032)

[Recursive bico function 5](#_Toc131620033)

[PYTHON PLOTS FOR RUNTIME 6](#_Toc131620034)

[Performance graphs 7](#_Toc131620035)

[Computing the average 10](#_Toc131620036)

[Comparison of Average Execution Time for Bico and Recursive Bico 11](#_Toc131620037)

[TIME COMPLEXITY ANALYSIS 12](#_Toc131620038)

[Time complexity of Iterative Bico 12](#_Toc131620039)

[Time complexity of recursive Bico 12](#_Toc131620040)

[Comparison 12](#_Toc131620041)

[HARDWARE, SOFTWARE & IDE 13](#_Toc131620042)

[CONCLUSION 13](#_Toc131620043)

## INTRO

The analysis of the time complexity of algorithms is a crucial aspect of computer science, particularly in the field of algorithm design and optimization. In this lab report, we aim to compare the time complexity of two functions, bico and recursiveBico, which compute binomial coefficients using different approaches. We will analyze the performance of these functions for different input sizes and evaluate their time complexity using the Big O notation. By examining the performance of these functions, we can gain a better understanding of the trade-offs between different algorithms and their efficiency in solving particular problems. Ultimately, this analysis can help us make more informed decisions when designing and optimizing algorithms for specific applications.

The functions, bico and recursiveBico, were implemented in C++ standard 11. The time complexity measurements were obtained and saved in a text file. The graphs displaying the performance of the functions were generated using the Python matplotlib.pyplot library. All of the measurements are in nanoseconds [ns].

## FUNCTIONS in C++ std=11

### Iterative Bico function

/\*\*

\* @brief Creates an iterative implementation of Pascal's triangle.

\* @param degree Degree of the polynomial.

\* @return A 2D array representing Pascal's triangle.

\* @remark The caller of this function is responsible for freeing the memory

\*/

int\*\* iterativePascal(int degree) {

int\*\* triangle = new int\*[degree+1];

for (auto j = 0; j <= degree; j++) {

triangle[j] = new int[j+1];

triangle[j][0] = 1;

for (auto k = 1; k < j; k++) {

triangle[j][k] = triangle[j-1][k-1] + triangle[j-1][k];

}

triangle[j][j] = 1;

}

return triangle;

}

/\*\*

\* @brief Computes a binomial coefficient using Pascal's triangle.

\* @param n The degree of the polynomial.

\* @param i The index of the coefficient.

\* @return The binomial coefficient.

\*/

int bico(int n, int i) {

int\*\* triangle = iterativePascal(n);

int result = triangle[n][i];

for (auto j = 0; j <= n; j++) {

delete[] triangle[j];

}

delete[] triangle;

return result;

}

#### Explanation

The two functions, iterativePascal and bico, work together to compute binomial coefficients using Pascal's triangle. iterativePascal creates a 2D array representing Pascal's triangle using an iterative approach. The outer loop iterates from 0 to the degree of the polynomial, and the inner loop computes each element of the triangle based on the previous row's elements. bico then calls iterativePascal to generate the triangle and uses the degree and index inputs to retrieve the desired binomial coefficient from the triangle. The memory allocated for the triangle is then freed to prevent memory leaks. The iterative approach of these functions allows for efficient computation of binomial coefficients and is a common method used in many programming languages.

### Recursive bico function

int recursiveBico(int degree, int index) {

if (index == 0 || index == degree) {

return 1;

}

else {

return recursiveBico(degree-1, index-1) + recursiveBico(degree-1, index);

}

}

/\*\*

\* @brief Creates a recursive implementation of Pascal's triangle.

\* @param degree Degree of the polynomial.

\* @return A 2D array representing Pascal's triangle.

\* @remark The caller of this function is responsible for freeing the memory

\*/

int\*\* recursivePascal(int degree) {

int\*\* triangle = new int\*[degree+1];

for (auto i = 0; i <= degree; i++) {

triangle[i] = new int[i+1];

for (auto j = 0; j <= i; j++) {

triangle[i][j] = recursiveBico(i, j);

}

}

return triangle;

}

#### Explanation

The second set of functions are recursive implementations of Pascal's triangle. The recursiveBico function computes a binomial coefficient using recursion, and the recursivePascal function generates Pascal's triangle using recursiveBico. The recursiveBico function checks whether the index is equal to zero or degree, and returns one in these cases. Otherwise, the function calls itself twice with decremented degree and index values, and adds their results to compute the binomial coefficient. The recursivePascal function initializes a 2D array and populates it using recursiveBico. Compared to the iterative functions, the recursive functions use less memory but are less efficient due to repeated function calls.

## PYTHON PLOTS FOR RUNTIME

#### EXPLAnation

The output of the program was stored into 2 text files Bico.txt and recursiveBico.txt. With the following lines of code in Python we can extract the data from those files (skipping the lines with a string) and appending them to a list containing 3 sub-lists for every measurement. Plots are titled accordingly.

List [Bico]

Sublists [(3,2), (10,5), (30, 15)]

List [recursiveBico]

Sublists [(3,2), (10,5), (30, 15)]

Text

Description automatically generated

### Performance graphs

#### CODE

Text

Description automatically generated

This code utilizes the Python library matplotlib.pyplot to create six plots depicting the performance of two functions: Bico and recursiveBico. The functions are implemented in C++ and are used to compute binomial coefficients. The first three plots display the execution times of the Bico function, with parameters (3,2), (10,5), and (30,15), respectively, in units of nanoseconds. The second set of three plots display the execution times of the recursiveBico function, also with parameters (3,2), (10,5), and (30,15), respectively. The plots are organized in three rows, each displaying a single plot, and are given a title corresponding to the function and parameters used. To avoid overlapping, the tight\_layout() method is used. Overall, these plots provide a clear visual representation of the relative performance of the Bico and recursiveBico functions across different input sizes.

#### GRPAPHS 1, 2 and 3

Line chart

Description automatically generated

#### GRAPHS 4, 5 and 6

Chart, line chart

Description automatically generated

### Computing the average

#### CODE

Text

Description automatically generated

This code imports two Python libraries, NumPy and Matplotlib. It then calculates the averages of the sub lists in Bico and recursiveBico using list comprehension and NumPy's mean() function. The next part of the code creates a bar chart that compares the average values of Bico and recursiveBico for three different pairs of arguments: (3,2), (10,5), and (30,15). The chart has two bars for each argument, one in blue for Bico and the other in green for recursiveBico. Matplotlib's bar() function is used to set the properties of each bar, and the set\_xticklabels() function sets the labels on the x-axis. The set\_ylim() function is used to set the y-axis limits so that the plot looks better, as the value of recursiveBico(30,15) is significantly larger than the other values. Finally, the chart is shown using Matplotlib's show() function after applying the tight\_layout() function to prevent overlapping of the axes labels. Overall, this code provides a clear visual representation of the performance differences between the iterative and recursive implementations of the binomial coefficient calculation function.

### Comparison of Average Execution Time for Bico and Recursive Bico

#### Graph

Chart, bar chart

Description automatically generated

The far right graph (green) is a lot bigger, but due to the size of the values I shrunk the scale down so all of the plots are visible.

## TIME COMPLEXITY ANALYSIS

### Time complexity of Iterative Bico

The time complexity of int bico(int n, int i) is O(). This is because the function iterativePascal() creates a 2D array of size , and the nested loops in that function iterate times to fill in the triangle. The subsequent loop in bico() iterates times to delete the 2D array. Therefore, the time complexity of iterativePascal(n) is O(), and the time complexity of bico() is also O() because it calls iterativePascal().

### Time complexity of recursive Bico

The time complexity of int recursiveBico(int degree, int index) is exponential, specifically O(), where n is the degree of the polynomial. This is because each recursive call branches into two more recursive calls, leading to a binary tree with nodes. Therefore, the total number of recursive calls made by the function is proportional to . This makes the recursive implementation of Pascal's triangle less efficient for greater amounts of data than the iterative implementation, which has a time complexity of O().

### Comparison

The time complexity of bico() and recursiveBico() functions differ significantly. The iterative approach of bico() function uses a precomputed Pascal's triangle to compute the binomial coefficient, taking constant time O(1) to return the coefficient. Therefore, the time complexity of bico() function is O(), where n is the degree of the polynomial. On the other hand, recursiveBico() function uses a recursive implementation of Pascal's triangle to compute the binomial coefficient. It recursively calls itself, computing the coefficient by summing the results of two recursive calls. Therefore, the time complexity of recursiveBico() function is O(), where n is the degree of the polynomial. This means that the recursiveBico() function is an exponential time algorithm, which makes it impractical for large values of n. Although the recursive approach has a simpler implementation, it is not suitable for large polynomial degrees. In contrast, the iterative approach used in bico() function provides a fast and efficient algorithm for computing binomial coefficients of moderate-sized polynomials.

## HARDWARE, SOFTWARE & IDE

#### HardWARE

* Device: 2022 Apple MacBook Pro
* Processor: Apple M2 chip with 8-core CPU (4 performance cores, 4 efficiency cores)
* Memory: 8 GB unified memory
* Storage: 512 GB solid-state drive (SSD)

#### SOFTWARE

* macOS Monterey v 12.5

#### IDE

* Visual Studio Code

## CONCLUSION

In conclusion, the time complexity of computing binomial coefficients using Pascal's triangle depends on the method used. While the iterative approach has a time complexity of O(), the recursive approach has an exponential time complexity of O(). Therefore, for large values of , the recursive approach can be significantly slower than the iterative approach.