TERM PROJECT REPORT

1. Introduction

In this project, we aimed to analyze the access log file from web server, focusing on HTTP requests. We have used hash functions and sorting algorithms for efficient data processing. Program finally prints the 10 most accesed resources with number of requests.

2. Components

2.1. hashTable.hpp

Public Methods

Constructor (hashTable) - Initializes the hash table with a given size (default is 25057).

Destructor (~hashTable) - For cleaning up resources, if necessary.

getHashVector - Returns the vector containing hash nodes.

getSize - Returns the size of the hash vector.

getElement - Overloaded methods to get a hash node by index, either as a mutable or constant reference.

Private Members

hashVector - A vector of hashNode which actually stores the hash table data.

pairedHash - A method to compute the hash value for a given string. The actual implementation is not provided in the header.

probe - A method for probing (handling collisions) in the hash table. Again, the implementation is not provided in this file.

2.2 Sorting.hpp

Implements sorting algorithms (quickSort, partition, and insertionSort) for sorting the hash table based on visit counts.

This header file, Sorting.h, defines a Sorting class, which provides static sorting functions to sort arrays (or more specifically, std::vectors) of hashNode objects. The hashNode struct is presumably defined in the termProject.hpp file. Let's break down the elements of the Sorting class:

Class Declaration: Sorting

The Sorting class encapsulates sorting algorithms. It is designed as a utility class, evident from the static methods, meaning that it doesn't need to be instantiated to use its functions.

Public Static Methods

quickSort:

Sorts a vector of hashNode elements using the QuickSort algorithm.

Parameters: A reference to a vector of hashNode (arr), and two integers (low and high) indicating the range of the vector to sort.

insertionSort (Overloaded):

Sorts a vector of hashNode elements using the Insertion Sort algorithm.

The first overload takes just the vector and sorts the entire vector.

The second, private overload is used to sort a specific range within the vector (from low to high).

Private Static Methods

swap:

A utility function to swap two hashNode elements.

Parameters: Pointers to two hashNode objects (a and b).

partition:

A helper function for quickSort.

It partitions the array into two parts based on a pivot element, so that elements less than the pivot are on the left, and elements greater than the pivot are on the right.

Parameters: A reference to a vector of hashNode and two integers indicating the range to partition.

insertionSort (Private Overload):

A helper function for sorting a subset of the vector using Insertion Sort.

Parameters: A reference to a vector of hashNode and two integers indicating the range to sort.

Usage:

Since the methods are static, they can be called without creating an instance of the Sorting class. For example,

```
Sorting::quickSort(myVector, 0, myVector.size() - 1);
```

would sort the entire myVector of hashNode elements using QuickSort.

2.3 Parallel Programming (Commented Out)

Since we have small data, it prolongs the time to give results so it is currently commented out. If we had larger data, using it with processChunk function could save us time.The function was likely intended to be used with a parallel programming framework (like OpenMP), but it's commented out due to performance issues. Here's a breakdown of its functionality:

Function Overview

Purpose: To process each line of a given string chunk, extract specific data, compute its hash value, and update the hash table (hashVector) accordingly.

Function Breakdown

Initializing Stream and Processing Lines

std::stringstream stream(chunk); : Converts the string chunk into a stream for easy line-by-line processing.

while (std::getline(stream, line)) { ... } : Iterates over each line in the chunk.

Extracting Data from Lines

The function looks for two specific characters ('\"' and 'H') to extract a substring from each line.

firstQuote and secondQuote store the positions of these characters.

If both characters are found (firstQuote != std::string::npos && secondQuote != std::string::npos), it extracts the substring request using substr.

Hashing and Probing

unsigned int hashValue = pairedHash(request); : Computes the hash value for the extracted string.

std::size_t index = probe(hashValue); : Finds an appropriate index in the hash table for this hash value using the probe function.

Handling Hash Table Updates

If a valid index is found (not equal to std::numeric_limits<std::size_t>::max()), the function proceeds to update the hash table.

The #pragma omp critical directive indicates a critical section, ensuring thread safety when multiple threads are updating the hash table concurrently.

Inside the critical section, the function checks if the computed index is already occupied.

If not occupied, it initializes the hash table entry with the new data (name, key, occupied, value).

If it is occupied (implying a hash collision or an existing entry), it simply increments the value for that entry.

Summary

processChunk is designed for parallel processing of data chunks in a hash table context. It reads and processes data line-by-line, extracts relevant substrings, hashes them, and updates the hash table. The function includes a critical section for thread safety, but it was commented out due to performance issues, possibly arising from the overhead of parallelization or contention in the critical section.

2.4 pairedHash Function

The pairedHash function is a method of the hashTable class in C++. This function is designed to generate a hash value for a given input string. Here's a breakdown of how it works:

Initialization of Hash Value: The function starts by initializing hashValue to 0. This variable will store the computed hash value.

Iterating Over the Input String: The function then iterates over each character (char c) in the input string (input).

Hash Calculation for Each Character: For each character in the string, the function performs the following operations:

hashValue << 3: This operation performs a left bit shift on hashValue by 3 places. Bit shifting hashValue by 3 places to the left is equivalent to multiplying hashValue by 8.

(hashValue << 3) - hashValue: After shifting, it subtracts hashValue from the result. This is effectively 8 * hashValue - hashValue, which simplifies to 7 * hashValue.

7 * hashValue + c: Finally, it adds the ASCII value of the current character c to the result. This step integrates the current character into the hash value.

These steps ensure that each character in the string affects the final hash value, and the use of bitwise operations and multiplication makes the computation quite efficient.

Return the Hash Value: After processing all characters in the string, the function returns the final hashValue.

This hashing method is relatively simple yet effective for a range of applications. The use of bit manipulation and arithmetic operations provides a good balance between speed and the distribution of hash values, which is desirable in a hashing function.

The three versions of the pairedHash function present different approaches to hashing a string. Let's compare them:

1. First Commented Version

This version processes two characters at a time from the input string.

Approach: For each pair of characters, it combines their ASCII values into a single std::size_t value. This is achieved by shifting the first character (c1) 8 bits to the left and then performing a bitwise OR with the second character (c2). This merged value is then combined with the current hash value using a combination of shifting and XOR operations.

2. Second Commented Version

This version iterates over each character of the string individually.

Approach: For each character, it shifts the current hash value to the left by 5 bits and to the right by 27 bits, then performs XOR operations with the character's ASCII value.

3. Un-Commented (Active) Version

This is the simplest of the three versions.

Approach: It multiplies the current hash value by 7 (using a combination of left shift and subtraction for efficiency) and adds the ASCII value of each character.

Summary

The first commented version provides a more complex and potentially more evenly distributed hashing mechanism, especially for longer strings, due to its pair-wise processing and bit manipulation.

The second commented version offers a mix of bit shifting and XOR operations, aiming for a balance between complexity and hash distribution.

The un-commented (active) version opts for simplicity and speed, suitable for general use cases but potentially less effective in avoiding collisions for a large or complex set of input strings.

In choosing a hash function, one must consider the specific requirements of the application, such as the expected size and type of input data, and the necessity to minimize hash collisions. The simpler method in the un-commented version might be adequate for many general applications, but the more complex methods could be more effective in scenarios where a more evenly distributed range of hash values is required.

2.5 getHash Function

The getHash function is a member of the hashTable class in C++. Its primary purpose is to read from a log file named access_log.txt, extract specific data from each line, compute a hash value for the extracted data, and store or update this information in a hash table. Here's a detailed explanation of its functionality:

Function Overview

Purpose: To read and process lines from a file, extracting specific strings (requests), computing their hash values, and updating the hash table (hashVector) with the frequency of each unique request.

Function Breakdown

Opening the File

`std::ifstream file("access_log.txt", std::ios::binary);` : Opens the log file in binary mode for reading. The binary mode is used here, possibly to handle any non-text data or to improve read performance by avoiding text mode translations.

`if (!file.is_open()) { ... }` : Checks if the file is successfully opened. If not, it prints an error message and returns from the function.

Reading the File in Chunks

`const std::size_t bufferSize = 16384;` : Sets a buffer size of 16,384 bytes (16 KB).

`std::vector<char> buffer(bufferSize);` : Creates a buffer to store chunks of data read from the file.

`std::string line, leftover;` : Declares strings to hold individual lines and any leftover data from the buffer.

`while (file.read(buffer.data(), bufferSize) || file.gcount() > 0) { ... }` : Reads the file in chunks. After each read, the loop checks if any data was read using file.gcount() and continues processing.

Processing Each Line

`std::stringstream stream(leftover + std::string(buffer.data(), bytesRead));` : Creates a stringstream combining any leftover data from the previous read and the current buffer content.

`leftover.clear();` : Clears the leftover string for the next iteration.

Inside the loop:

`while (std::getline(stream, line)) { ... }` : Reads lines from the stringstream.

Extracting and Hashing Data

`std::size_t firstQuote = line.find('\"'); and std::size_t secondQuote = line.find('H', firstQuote + 1);` : Finds the positions of specific characters (a quote and 'H') in each line. These positions are used to extract a substring.

std::string request = line.substr(firstQuote + 5, secondQuote - firstQuote - 6); : Extracts the substring based on the found positions.

unsigned int hashValue = pairedHash(request); : Computes the hash value for the extracted string using the pairedHash function.

std::size_t index = probe(hashValue); : Finds an appropriate index in the hash table for this hash value.

Updating the Hash Table

Checks if the computed index is valid (not the max value for std::size_t).

If valid, it either initializes a new entry in the hash table or increments the value (frequency count) of an existing entry.

Closing the File

file.close(); : Closes the file after processing is complete.

Summary

The getHash function is responsible for reading data from a log file, extracting relevant parts of each line, hashing these parts, and updating a hash table with this information. This function is crucial for analyzing and storing the frequency of specific requests found in the log file. It demonstrates file reading, string manipulation, and the application of a hash table for frequency analysis.

2.6 Main Function

i. Acts as the program's entry point.

ii. Creates a hashTable instance, populates and sorts the hash table, and prints the top records.

iii. Measures and displays the program's runtime.

iv. Workflow Overview

2.7 Probe Function

The probe function is a method within the hashTable class in C++, designed to handle collisions in the hash table. It uses quadratic probing to find a suitable index in the hash table for a given key. Here's a detailed explanation of the function:

Function Signature

std::size_t hashTable::probe(int key) : The function is a member of the hashTable class. It takes an integer key as its parameter and returns a std::size_t, which is the index in the hash table where the key can be inserted or found.

Function Body

Initialization

const std::size_t maxAttempts = hashVector.size();: The maximum number of probing attempts is set to the size of the hash table (hashVector). This prevents the function from entering an infinite loop if the table is full.

std::size_t i = 0;: Initializes a counter i to zero, which will be used in the probing process.

Probing Loop

while (i < maxAttempts) { ... }: A loop that continues as long as the number of attempts is less than maxAttempts.

Inside the loop:

std::size_t index = (key + i * i) % hashVector.size(); : Calculates the index at which to try inserting or finding the key. It uses quadratic probing, where the probe step size increases quadratically (i * i). This step size is added to the original key, and the result is modulo'd with the size of the hash table to ensure it remains within valid bounds.

if (!hashVector[index].occupied || hashVector[index].key == key) { : This condition checks if the calculated index is suitable for inserting the key. There are two cases:

!hashVector[index].occupied: The index is not occupied, meaning it's free to be used.

hashVector[index].key == key: The index is already occupied by the same key (useful for operations like searching or updating values associated with this key).

return index;: If either condition is met, the function returns the current index as the suitable location.

i++; : Increment the counter i for the next probing step if the current index is not suitable.

Failure to Find a Slot

return std::numeric_limits<std::size_t>::max(); : If the function exhausts maxAttempts without finding a suitable index, it returns a special value (std::numeric_limits<std::size_t>::max()) indicating failure. This value is effectively used as an error code, signaling that there is no available slot for the key in the hash table.

Summary

The probe function implements quadratic probing to resolve collisions in the hash table. Quadratic probing helps in evenly distributing keys across the hash table, reducing clustering issues common in linear probing.

However, if the hash table is close to being full, this method may struggle to find free slots, leading to performance degradation. Therefore, it's important to maintain a reasonable load factor (the ratio of the number of elements to the size of the hash table) for efficient performance.On the other hand, if the data is not provided previously, the re-hash and re-size functions ,which are commented out, should be used to gather data.

2.8 resizeAndRehash Function

The commented resizeAndRehash function in the code is intended to resize and rehash the hashVector in the hashTable class. This function is crucial in scenarios where the hash table needs to grow in size to accommodate more elements or to maintain a good performance by keeping the load factor (the ratio of the number of stored elements to the table size) within an optimal range. Here's a detailed explanation of its functionality:

Overview

The function aims to increase the size of the hashVector and rehash all existing elements into this new, larger vector. This process is necessary because changing the size of a hash table invalidates the previous hash values, requiring all elements to be rehashed according to the new table size.

Function Breakdown

Checking if More Sizes are Available

The function first checks if there are more sizes to grow into using currentHashSizeIndex and an array hashsize (not shown in the code snippet). This array seems to contain predefined sizes for the hash table.

If currentHashSizeIndex is at its maximum (i.e., the hash table has reached its largest predefined size), the function prints a message and returns, indicating that no further resizing is possible.

Calculating New Size and Creating New Hash Vector

int newSize = hashsize[++currentHashSizeIndex]; : The size of the hash table is increased to the next value in the hashsize array.

std::vector<hashNode> newHashVector(newSize); : A new hash vector newHashVector is created with the increased size.

Rehashing Elements

The function then iterates through each element (node) in the original hashVector.

For each occupied node (if (node.occupied)), it calculates a new index in the resized hash table using the probe function. The modulo operation (% newSize) ensures that the index is within the bounds of the new table size.

The node is then placed at this new index in newHashVector.

Replacing the Old Hash Table

Finally, the original hashVector is replaced with newHashVector using the std::move operation. This effectively transfers the contents of newHashVector to hashVector, and the old hashVector is now empty. The use of std::move is a performance optimization, as it avoids copying all elements and instead transfers ownership of the resources.

Summary

The resizeAndRehash function is a critical component for dynamically resizing a hash table. This functionality is key to maintaining the efficiency of hash table operations, especially as the number of elements grows. By resizing and rehashing, the hash table can continue to operate effectively without excessive collision or performance degradation. However, it's important to note that resizing and rehashing can be a computationally expensive operation, as it involves re-computing the hash for every element and can momentarily require additional memory to hold two versions of the hash table.

2.9 Sorting.cpp

This C++ code defines the implementation of sorting algorithms within the Sorting class, specifically designed for sorting a std::vector of hashNode objects. Let's go through each function:

swap

Purpose: Swaps two hashNode elements.

Parameters: Pointers to two hashNode objects (a and b).

Process: It temporarily saves the value of a in t, assigns the value of b to a, and then assigns the value stored in t to b. This effectively swaps the values of a and b.

insertionSort

Purpose: Performs the Insertion Sort algorithm on a subrange of the vector.

Parameters: A reference to a vector of hashNode (arr) and two integers (low and high) indicating the range to sort.

Process: It iterates from the second element of the specified range to the end. For each element, it shifts all larger elements in the sorted part of the array (left side) one position to the right to create a space for the current element, which is then inserted into its correct position.

partition

Purpose: Used in the QuickSort algorithm to partition the array.

Parameters: A reference to a vector of hashNode and two integers indicating the range to partition.

Process: It selects the last element as the pivot and rearranges the array so that elements smaller than the pivot are on the left, and elements larger are on the right. It then swaps the pivot into its correct position and returns the index of the pivot.

quickSort

Purpose: Sorts a vector of hashNode elements using a hybrid of QuickSort and Insertion Sort algorithms.

Parameters: A reference to a vector of hashNode and two integers indicating the range to sort.

Process: If the size of the range is less than or equal to 20, it uses Insertion Sort for more efficiency in small arrays.

For larger ranges, it uses QuickSort:

The array is partitioned using the partition method.

QuickSort is then recursively applied to the subarrays before and after the partition.

Notably, in this implementation, there is a commented section suggesting an intention to conditionally apply sorting to the elements before the partition only if the segment size is less than 20. However, this part is missing, and QuickSort is used unconditionally for the left subarray.

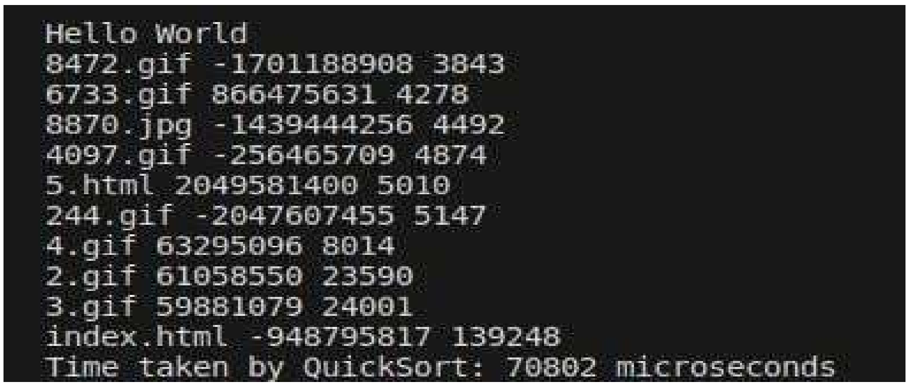
This implementation provides an efficient way to sort hashNode objects, leveraging the speed of QuickSort for larger arrays while optimizing smaller subarrays with Insertion Sort. The use of Insertion Sort for small segments is a common optimization in QuickSort implementations, as QuickSort's overhead is not justified for small arrays.

I’m using a modified version of quicksort which is running on O(n) time complexity for the firts 20 elements. Time complexity is proven by equation below. After those elements because I have to also do low part, I switched to insertion sort because its running time is lower with smaller data although time complexity is higher.

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} \ldots = \sum_{i=1}^{x} \frac{n}{2^{n-1}}$$

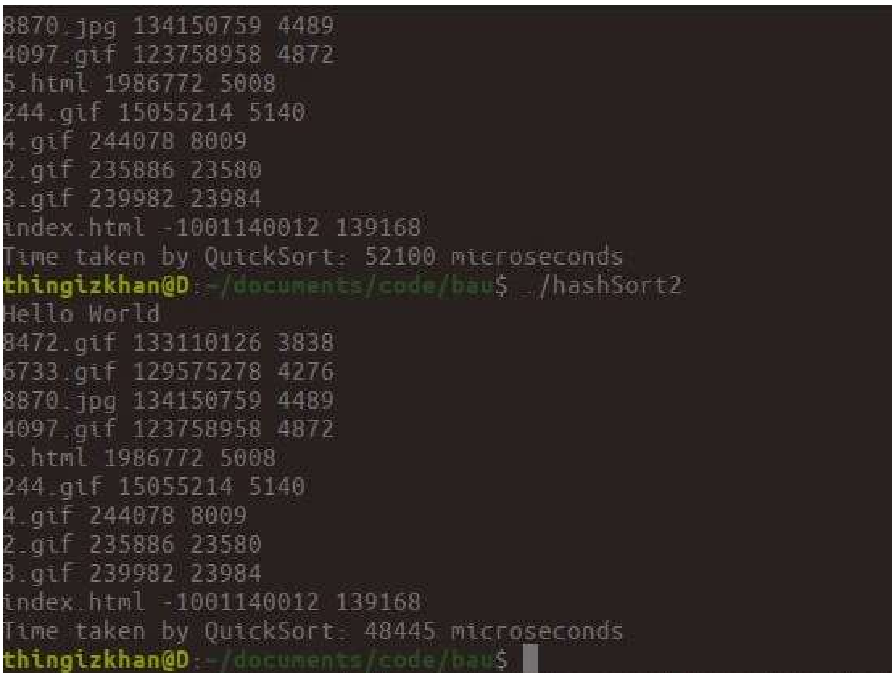
$$2n = \sum_{i=1}^{x} \frac{n}{2^{n-1}}$$

3. Comparisons



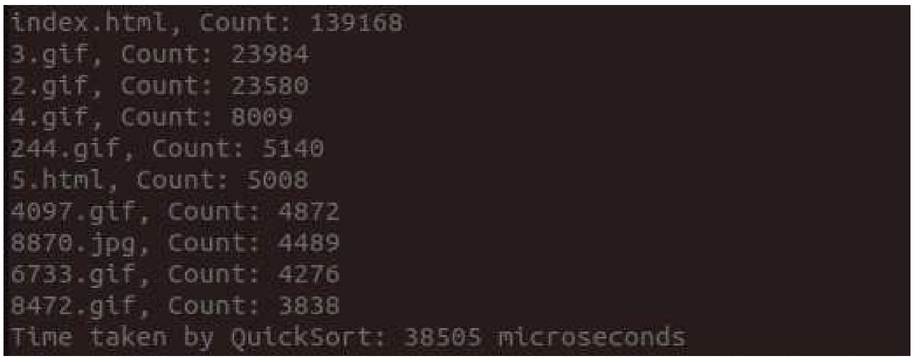
```
Hello World
8472.gif -1701188908 3843
6733.gif 866475631 4278
8870.jpg -1439444256 4492
4097.gif -256465709 4874
5.html 2049581400 5010
244.gif -2047607455 5147
4.gif 63295096 8014
2.gif 61058550 23590
3.gif 59881079 24001
index.html -948795817 139248
Time taken by QuickSort: 70802 microseconds
```

Reverse sorted top 10 with VS Code compiler 71ms



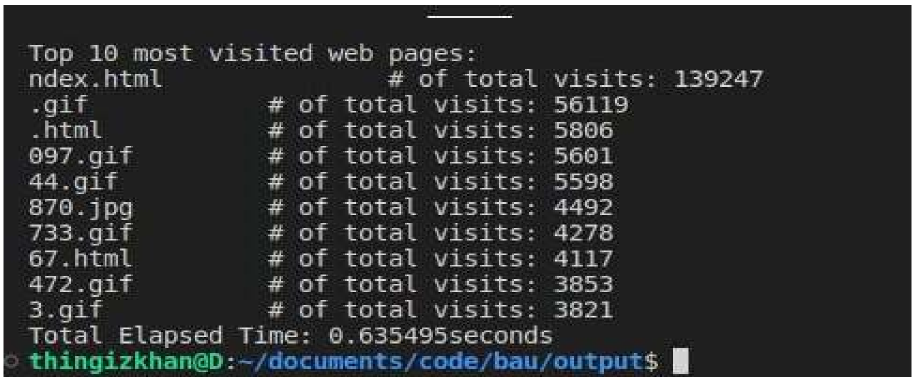
```
8870.jpg 134150759 4489
4097.gif 123758958 4872
5.html 1986772 5008
244.gif 15055214 5140
4.gif 244078 8009
2.gif 235886 23580
3.gif 239982 23984
index.html -1001140012 139168
Time taken by QuickSort: 52100 microseconds
thingizkhan@D:~/documents/code/bau$ ./hashSort2
Hello World
8472.gif 133110126 3838
6733.gif 129575278 4276
8870.jpg 134150759 4489
4097.gif 123758958 4872
5.html 1986772 5008
244.gif 15055214 5140
4.gif 244078 8009
2.gif 235886 23580
3.gif 239982 23984
index.html -1001140012 139168
Time taken by QuickSort: 48445 microseconds
thingizkhan@D:~/documents/code/bau$
```

Reverse sorted top 10 with GCC 48ms



```
index.html, Count: 139168
3.gif, Count: 23984
2.gif, Count: 23580
4.gif, Count: 8009
244.gif, Count: 5140
5.html, Count: 5008
4097.gif, Count: 4872
8870.jpg, Count: 4489
6733.gif, Count: 4276
8472.gif, Count: 3838
Time taken by QuickSort: 38505 microseconds
```

Sorted top 10 with GCC and O3 optimizer 38ms



```
Top 10 most visited web pages:
ndex.html               # of total visits: 139247
.gif            # of total visits: 56119
.html           # of total visits: 5806
097.gif         # of total visits: 5601
44.gif          # of total visits: 5598
870.jpg         # of total visits: 4492
733.gif         # of total visits: 4278
67.html         # of total visits: 4117
472.gif         # of total visits: 3853
3.gif           # of total visits: 3821
Total Elapsed Time: 0.635495seconds
thingizkhan@D:~/documents/code/bau/output$
```

Unordered map VS Code compiler 635ms

4. Compiler Choice

We chose GCC compiler:

The -O3 optimization level in GCC (GNU Compiler Collection) is one of several optimization levels you can choose when compiling C or C++ code. Each optimization level instructs the compiler on how aggressively it should optimize the code. Here's a more detailed look at what -O3 entails:

Level of Optimization: -O3 is one of the highest levels of optimization provided by GCC. It includes all the optimizations of -O2 and adds further optimizations.

What It Does:

Aggressive Inlining: Functions are more aggressively inlined at this level, meaning the compiler replaces function calls with the actual function code. This can increase execution speed but may result in a larger binary.

Loop Unrolling and Vectorization: Loops may be unrolled (i.e., repetitive code is expanded to reduce loop overhead) and vectorized (using vector instructions of CPUs for parallel processing within a single CPU instruction).

Advanced Optimizations: It enables more complex optimizations, like prefetching, scalar replacement, and others that can significantly improve performance for some types of calculations and algorithms.

Trade-offs:

Increased Compile Time: The compilation process may take longer because the compiler performs more analysis and transformations.

Larger Binary Size: The executable size is often larger due to inlining and other code expansions.

Possible Instability: In rare cases, the aggressive optimizations might introduce bugs or instabilities, especially in complex software.

For compile the code i used this prompt:

```
g++ -O3 -Wall -Wextra -std=c++17 -flto termProject.cpp sorting.cpp -o termProject
```

AZRA ÇELİK 2102696

MEHMET CENGİZHAN KINAY 2102805