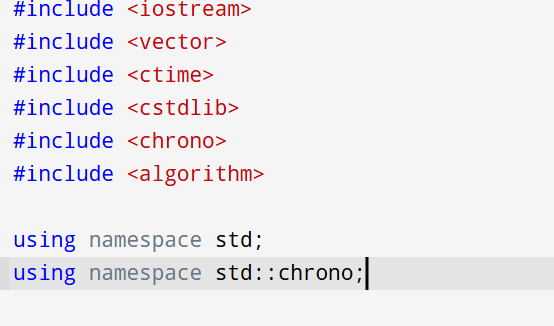
Header Files and Namespace Declaration



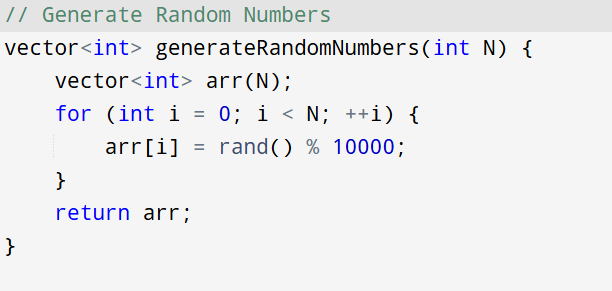
This section includes the necessary libraries that provide the building blocks for the program, along with namespace declarations that simplify code syntax. Each header file serves a specific purpose:

* <iostream> is used for input/output operations via cin and cout.
* <vector> allows the use of dynamic arrays, which are essential for storing and manipulating lists of numbers.
* <ctime> and <cstdlib> enable time based seeding of the random number generator, ensuring that random values differ in each execution.
* <chrono> is used for high-resolution time measurement to evaluate algorithm performance.
* <algorithm> provides access to STL utilities like sort().

The using namespace std; and using namespace std::chrono; lines allow direct access to standard and time-related functions without the need to prepend std:: or chrono::.

The using namespace std; and using namespace std::<bredm overall, the lines are stated that enable the direct access to the standard and time-related functions without having to prepend it with std: or chrono:

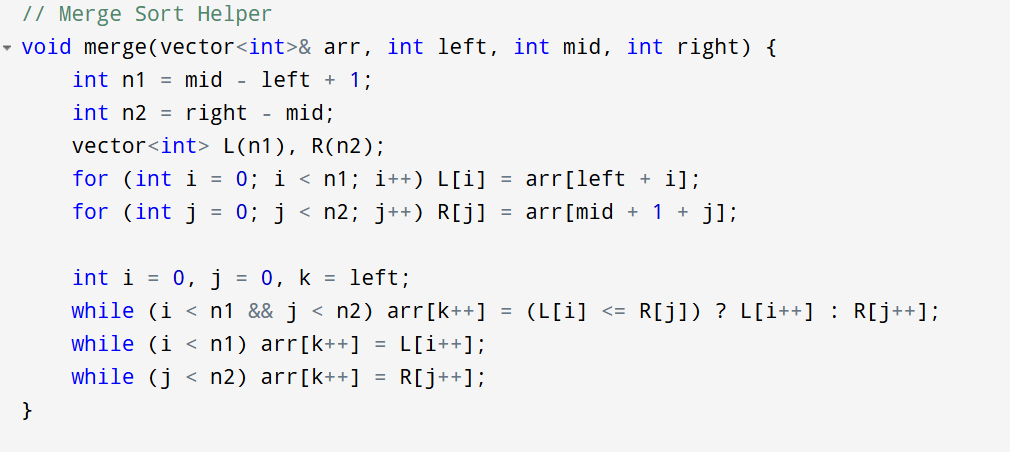
Random Number Generator Function



This part of the code is responsible for creating a list of random numbers that the rest of the program will work with. The function generateRandomNumbers(int N) takes an input N, which is the number of elements the user wants, and creates a vector filled with N random integers ranging from 0 to 9999. Each number is generated using rand() % 10000, which ensures variety in the data every time the program runs.

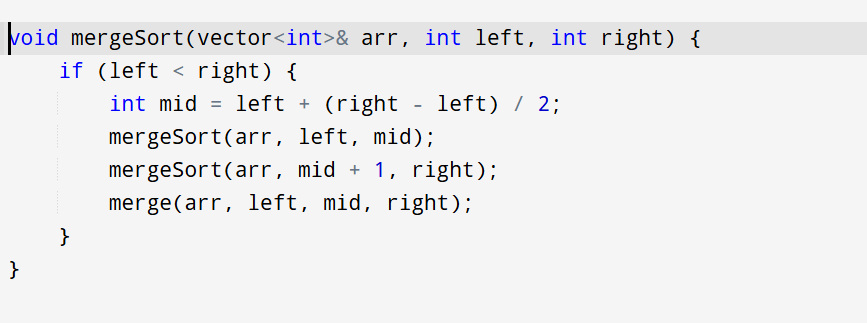
It gives the program a fresh set of numbers on each run, making it more dynamic and useful when comparing the speed and accuracy of sorting and searching techniques

Merge Function for Merge Sort

This function is the heart of the Merge Sort algorithm—it’s where the actual sorting happens after the array has been divided. The merge() function takes a portion of the array, which has already been split into two sorted halves, and merges them into a single sorted segment. It does this by copying the left and right parts into temporary vectors L and R, and then comparing their elements one by one to rebuild the original array in order.

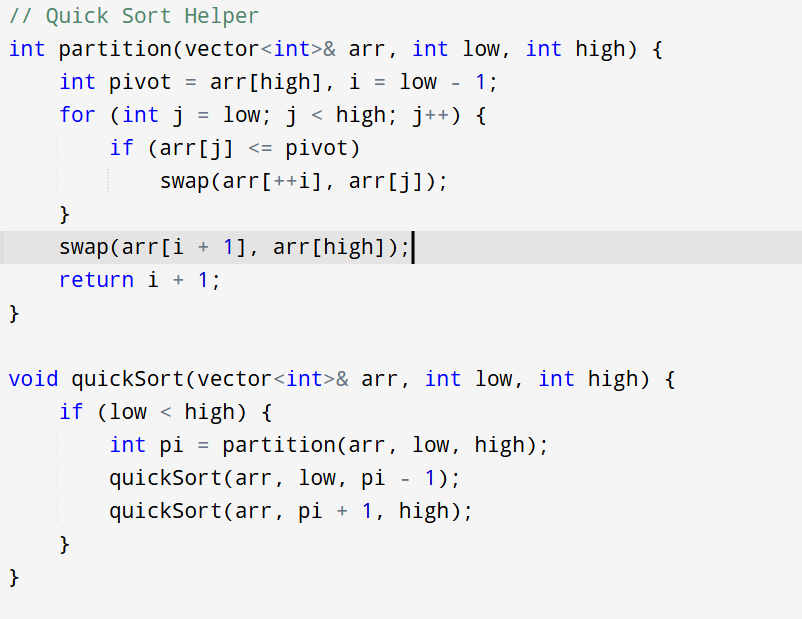
The loop continues to pick the smaller value from either the left or right half and places it back into the original array. Once one side runs out of elements, the rest of the elements from the other side are copied over directly, since they are already sorted. This approach ensures that the merged section remains sorted.

Recursive Division with mergeSort



The mergeSort function embodies the **divide-and-conquer** strategy by recursively splitting the array into ever smaller segments until they’re trivially sorted (i.e., contain one element), then stitching them back together in order. It calculates the midpoint safely with left + (right – left) / 2—which avoids potential integer overflow and recursively sorts the left half [left…mid] and the right half [mid+1…right] before calling merge() to combine them into a single sorted run.

Quick Sort and Partitioning Logic



This section implements the **Quick Sort algorithm**, a popular and efficient sorting method. It consists of two functions: partition() and quickSort().

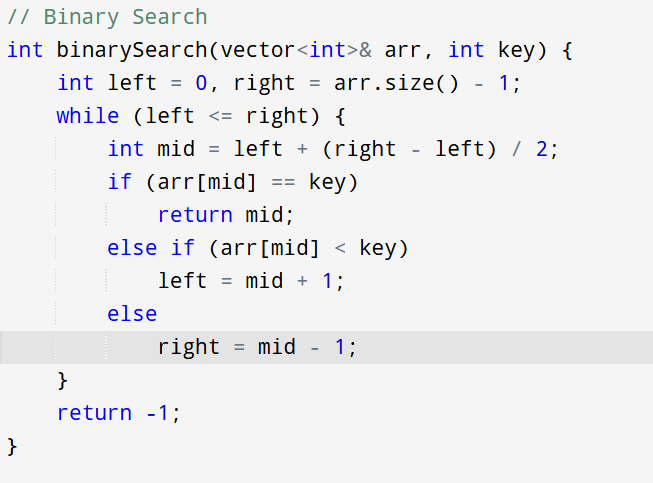
The partition() function selects the last element in the array as a **pivot** and rearranges the array so that:

* All elements smaller than or equal to the pivot are placed to its left.
* All elements greater than the pivot are placed to its right.

To do this, it uses a pointer i to track the boundary between elements less than or equal to the pivot. As it loops through the array, it swaps elements as needed. Once all elements are checked, it swaps the pivot into its correct sorted position and returns the index where the pivot ended up.

The quickSort() function uses recursion to sort the two halves of the array, divided by the position of the pivot returned by partition(). It keeps splitting and sorting the subarrays until the entire array is sorted.

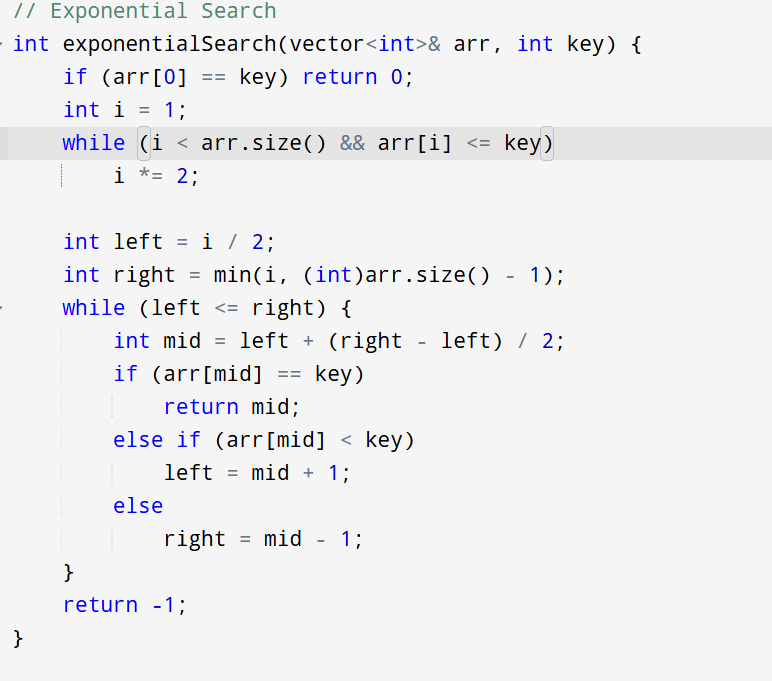
Binary Search Function



This section contains the classic **Binary Search** algorithm, designed for efficiently finding an element in a **sorted array**. The function takes a vector arr and a search key, then repeatedly divides the search range in half to narrow down the location of the key.

The logic begins by setting two pointers left at the start of the array and right at the end. In each iteration of the loop, it calculates the mid index and compares arr[mid] with the key. If the key matches, it returns the index. If the key is smaller, it narrows the search to the left half; if larger, it searches the right half. This continues until the element is found or the search range becomes invalid (left > right), in which case it returns -1 to indicate the key was not found.

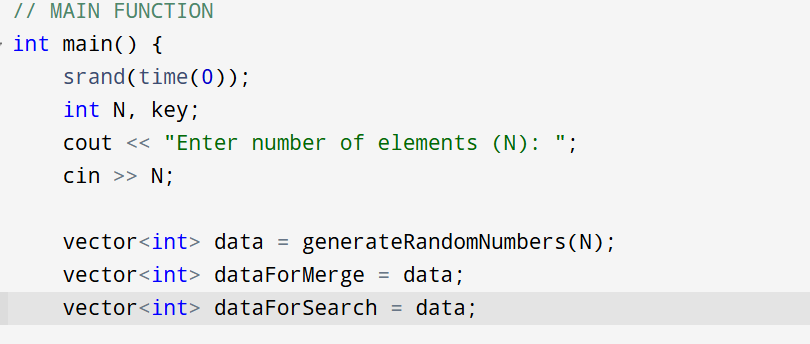
Exponential Search Function



This section implements **Exponential Search**, a hybrid search algorithm that combines fast range detection with efficient binary search. It is particularly useful when searching in large, sorted arrays, especially when the element is likely to be closer to the beginning.

The function starts by checking if the first element is the key. If not, it enters a loop that **doubles the index i** (i \*= 2) until it either exceeds the size of the array or finds a value greater than the key. This quickly identifies a subrange [i/2 ... min(i, n-1)] where the key might exist. Once the range is found, it uses **Binary Search** within that specific range to locate the key.

Main Function



This part of the main() function sets up the initial environment for the program. It begins by seeding the random number generator with the current system time using srand(time(0)), ensuring that each program run produces different sets of random numbers.

The user is then prompted to input the number of elements (N) they want to generate. Using this value, the program creates a vector data filled with random integers via the generateRandomNumbers() function. To allow fair and isolated comparison of sorting and searching algorithms, two copies of this original dataset are made:

* dataForMerge for Merge Sort
* dataForSearch for Binary and Exponential Search

Sorting Execution and Timing Measurement

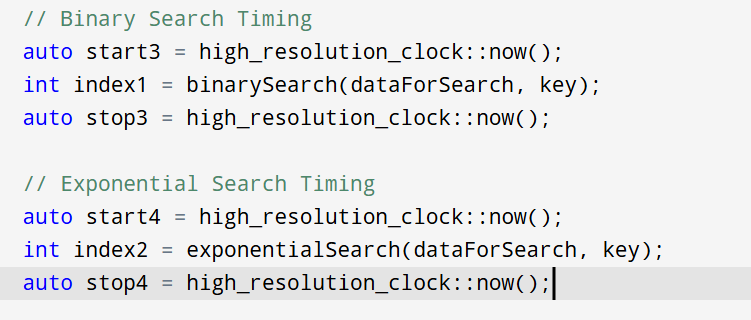


This section executes both **Quick Sort** and **Merge Sort** on separate copies of the same dataset and records how long each algorithm takes. It uses high\_resolution\_clock::now() from the <chrono> library to capture precise timestamps before and after sorting, and calculates the duration in milliseconds using duration\_cast<milliseconds>().

* The quickSort() function is called on data, and its time is printed immediately after.
* Then, mergeSort() is applied to dataForMerge, followed by its timing output.
* Lastly, dataForSearch is sorted using the built-in sort() function so it's ready for Binary and Exponential Search operations.

After the sorting benchmarks, the user is prompted to input the number they want to search for in the sorted dataset.

Searching and Performance Timing

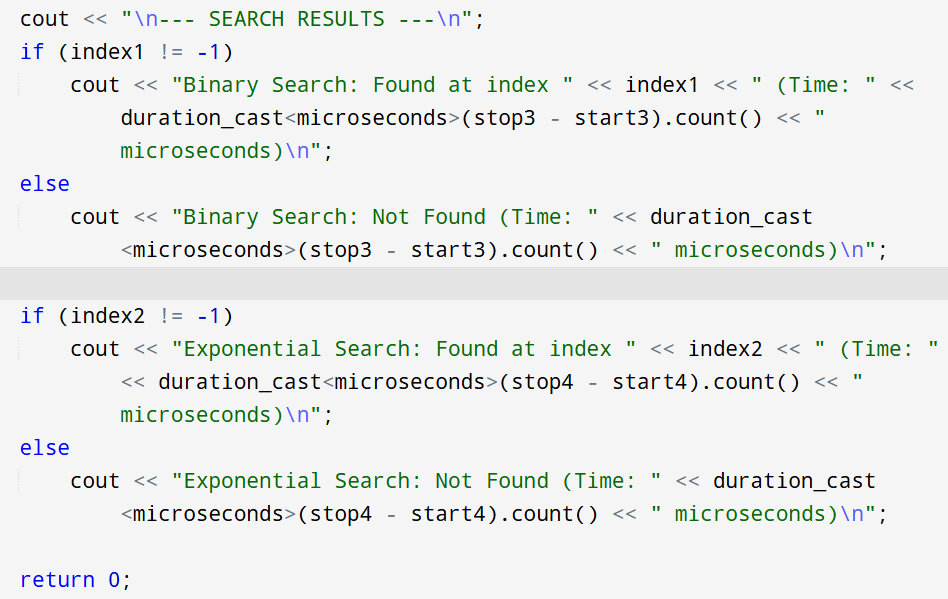


This part of the main() function focuses on executing the **Binary Search** and **Exponential Search** algorithms, while measuring and comparing how long each one takes. It uses high\_resolution\_clock to track execution time with microsecond-level precision.

First, it times the Binary Search function by capturing the time before and after calling binarySearch() on the already sorted dataForSearch. The same process is repeated for exponentialSearch() using the same dataset and search key.

Both functions return the index of the element if found, or -1 if not found. These results are stored in index1 and index2, respectively, for display later.

Displaying Search Results



In this final section of the program, the results of both search algorithms are printed to the user in a clean and readable format. A header --- SEARCH RESULTS --- is printed for clarity. Then, for both **Binary Search** and **Exponential Search**, the program checks if the returned index is not -1 (meaning the key was found). If the key is found, it prints the index along with how many microseconds the search took. If not found, it prints a "Not Found" message along with the timing.

This design ensures that the user can clearly see:

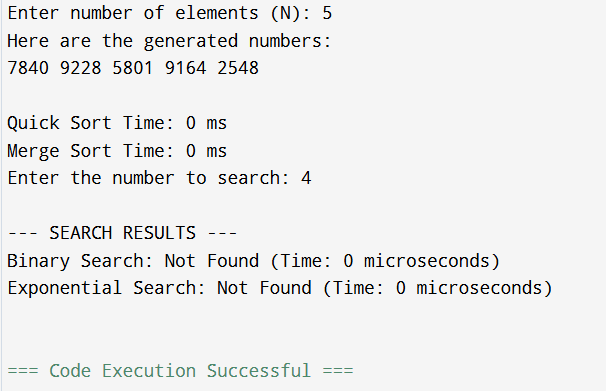
1. Whether each algorithm found the number.
2. How fast each one was in doing so.

Finally, the main() function ends with a return 0;, signaling that the program has completed successfully.

Examples of Code Execution:

A screenshot of a computer program

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

