Data Structures and Algorithms

Chapter 1: Time complexity, Arrays, Pointers

1. Time Complexity

Time complexity refers to the measure of how the runtime of an algorithm or a piece of code increases as the size of the input increases. It quantifies the **amount of time an algorithm takes to run as a function of the input size.** It measures the number of times a particular instruction or operation is executed by an algorithm, rather than the actual time taken to execute the algorithm. Time complexity is expressed using **Big O** notation, which provides an **upper bound** on the growth rate of the algorithm's runtime as the input size increases.

Big O Concepts:

Growth is with respect to input

• **O(1)** - Constant time: The runtime of the algorithm remains constant regardless of the input size.

Example:

```cpp
#include <iostream>
#include <string>
int main() {
        int arr[]{ 1,2,3,4,5 };
        int index{ 2 };

        int element{ arr[index] };
        std::cout << "Element at index " << index << ": " << element << '\n';
        return 0;
}
```

The code above initializes an array arr with values {1, 2, 3, 4, 5} and an integer variable index with the value 2. It then retrieves the element at the specified index from the array and assigns it to the variable element. Finally, it prints the index and the element using std::cout. The time complexity of this code is O(1) because it performs a constant number of operations, regardless of the size of the array.

If you edit the code to increase the number of elements in the array arr, the time complexity of the code will NOT change.

```cpp
#include <iostream>
#include <string>
int main() {
        int arr[100]{};
        int index{ 2 };
        int element{ arr[index] };
        std::cout << "Element at index " << index << ": " << element << '\n';
        return 0;
}
```

Even though the array size is now 100, the code still performs the same number of operations as before and takes the same constant amount of time to access and print an element.

• **O(log n)** - Logarithmic time: The runtime of the algorithm increases logarithmically with the input size.

Example: binary search in a sorted array.

• O(n) - Linear time: The runtime of the algorithm increases linearly with the input size.

Example:

```cpp
#include <iostream>
void printArray(int arr[], int size) {
        for (int i{ 0 }; i < size; ++i) {
                std::cout << arr[i] << " ";
        }
}

int main() {
        int arr[]{1,2,3,4,5};
        int size{ std::size(arr) };

        printArray(arr, size);
        return 0;
}
```

The code above prints the elements of an array. It defines a function called printArray that takes an array arr and its size as parameters. The function uses a for loop to iterate over the elements of the array and prints each element using the std::cout statement. The main function initializes an array arr with values 1, 2, 3, 4, and 5, and then calls the printArray() function to print the elements of the array. Finally, the main function returns 0, indicating successful execution. The time complexity of this code is O(n), where n is the size of the array, because the printArray function iterates over each element of the array once. The time it takes to execute the code is directly proportional to the size of the array.

• O(n log n) - Linearithmic time: The runtime of the algorithm increases linearly multiplied by the logarithm of the input size.

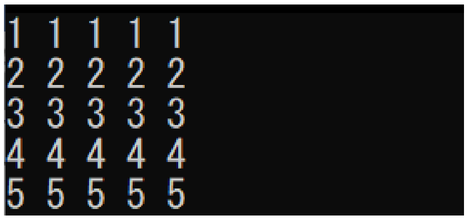
Example: sorting algorithms like merge sort and quicksort.

• O(n^2) - Quadratic time: The runtime of the algorithm increases quadratically with the input size.

Example:

```cpp
#include <iostream>
void printArrayElements(int arr[], int size) {
        for (int i{ 0 }; i < size; ++i) {
                for (int j{ 0 }; j < size; ++j) {
                        std::cout << arr[i] << " ";
                }
                std::cout << '\n';
        }
}

int main() {
        int arr[]{1,2,3,4,5};
        int size{ std::size(arr) };

        printArrayElements(arr, size);
        return 0;
}
```

Output:



```
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

The code above prints all the elements of an array in a specific pattern. The printArrayElements function takes an array arr and its size as parameters. It uses nested loops to iterate over the array elements and print them. The outer loop iterates over the array elements from index 0 to size-1, and the inner loop also iterates over the array elements from index 0 to size-1 (size-1 because array indices in C++ start from 0). As a result, each element of the array is printed size times. Therefore, the time complexity of this code is O(n^2), where n is the size of the array.

• O(2^n) - Exponential time: The runtime of the algorithm grows exponentially with the input size.

Example: generating all subsets of a set.

• O(n!) - Factorial time: The running time of an algorithm increases factorially with the size of the input.

Example: Using a recursive function to generate all permutations of a given set of elements

Constants are dropped

In big O notation, constants are dropped because they represent the scaling factor of the growth of a function. The purpose of big O notation is to provide an upper bound on the growth rate of a function as the input size increases towards infinity. Dropping the constants allows us to focus on the dominant term or the highest order of growth, which provides a more general and simplified understanding of the algorithm's efficiency.

When analyzing the complexity of an algorithm using big O notation, dropping constants simplifies the analysis by abstracting away specific implementation details. The main idea is that **constant factors become less significant as the input size becomes larger.** The primary goal of big O notation is to assess the algorithm's scalability and how it performs in the long run, which is why the focus is on the growth rate rather than specific constant values.

N = 1, O(10N) = 10, O(N^2) =1

N = 5, O(10N) = 50, O(N^2) = 25

N = 100, O(10N) = 1,000, O(N^2) = 10,000 // 10X bigger

N = 1000, O(10N) = 10,000, O(N^2) = 1,000,000 // 100x bigger

N = 10000, O(10N) = 100,000, O(N^2) = 100,000,000 // 1000X bigger

Theoretical vs practical difference

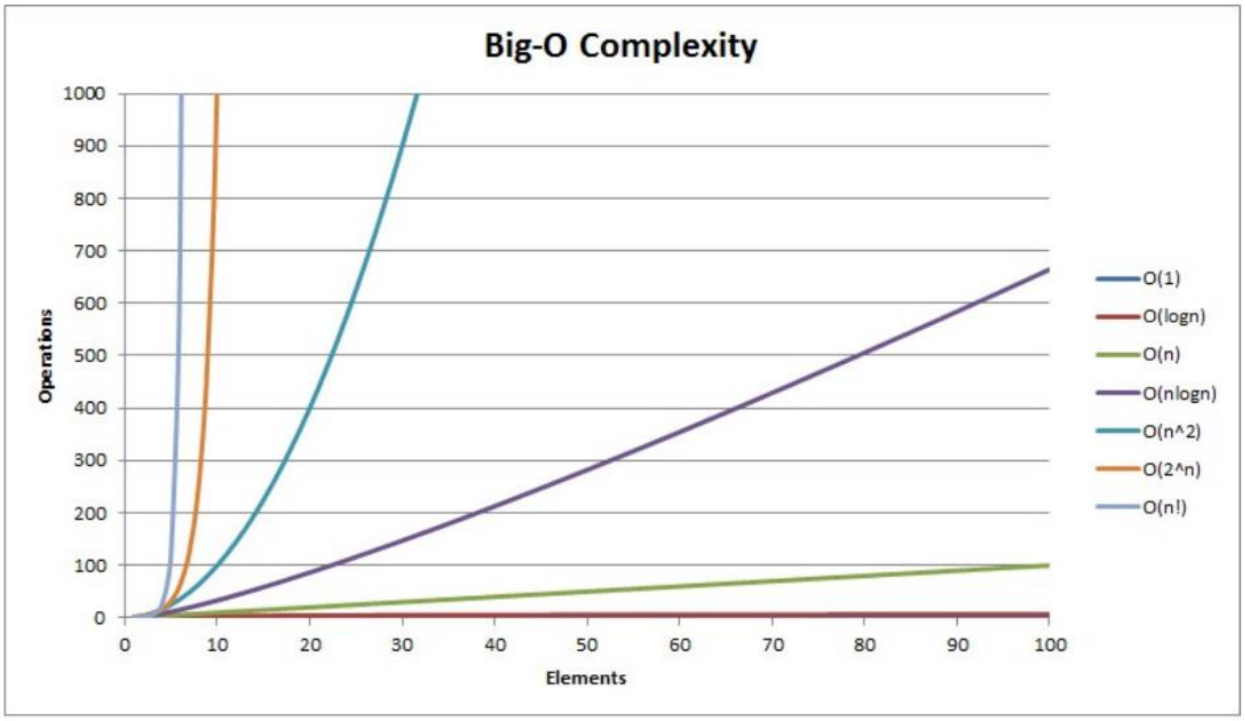
It's important to note that dropping constants doesn't mean constant factors are always irrelevant. In some situations, constant factors can have a practical impact on the algorithm's performance.

While constants are dropped in Big O notation, they can have a significant impact on the actual running time of an algorithm for small input sizes or in specific scenarios.

Just because N is faster than N^2, doesn't mean that its always faster for smaller input. O(100N) is faster than O(N^2) but practically, n^2 can probably be a little bit faster for smaller inputs.

We care about the worst-case scenario

The worst-case scenario represents the maximum amount of time or space an algorithm may require for any given input size



2. Arrays

Arrays in C++ have a fixed size, which is determined at compile-time. They can store elements of any data type, including built-in types (int, float, etc.) and user-defined types (structs, classes).

Syntax:

```
type arrayName[arraySize]{initializer list};
```

Example:

```
int arr1[5]{}; // arr1 is now {0,0,0,0,0}
```

The above syntax is known as zero initialization and is available in C++11 and later. It ensures that all elements of the array are initialized to their default values, which is 0 for int.

Another syntax:

```
int arr2[5]{1,2,3,4,5};
int arr3[]{ 1,2,3,4,5 };
```

In arr2, **we are explicitly specifying the size** of the array as 5. This means that the array will have exactly 5 elements, and each element will be initialized with the corresponding value in the initializer list.

In arr3, the **size of the array is automatically determined** based on the number of elements in the initializer list. In this case, the array will have a size of 5 because there are 5 elements in the initializer list. This syntax is known as "array size deduction" and was introduced in C++17.

It's important to note that in both cases, the **array elements are initialized in the order they appear in the initializer list**. If the initializer list has fewer elements than the size of the array, the remaining elements will be value-initialized (which means they will be set to their default values).

Example:

```
int arr2[5]{1,2,3,4,5}; // arr2 has elements 1,2,3,4,5
int arr4[5]{ 1,3,5 };   // arr4 has elements 1,3,5,0,0
```

To access elements in an array, you can use the subscript operator [] along with the index of the element you want to access. ***Note that the index of arrays starts with 0.***

```
int arr[5]{9,8,3,1,4};
int secondElement{ arr[1] };    // secondElement's value is 8
std::cout << arr[3];            // will print 1
```

You can use a variable as the index of the element you want to access:

```
int arr[]{74,16,98,52,63};
int x{ 3 };
int y{ 1 };
std::cout << arr[x] << '\n'; // will print 52
std::cout << arr[y] << '\n'; // will print 16
```

To check the number of elements in an array, we have a few options:

• Using the sizeof operator: We can use the sizeof operator to determine the size of an array in bytes. Divide the total size by the size of a single element to get the number of elements in the array.

```
#include <iostream>

int main() {
        int arr[]{1,2,3,4,5};
        int size{ sizeof(arr)/sizeof(arr[0])};

        std::cout << "The size of the array is: " << size << '\n';
        return 0;
}
```

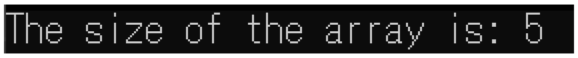
• Using std::size:

```
#include <iostream>
```

```cpp
int main() {
        int arr[]{1,2,3,4,5};
        int size{ std::size(arr)};

        std::cout << "The size of the array is: " << size << '\n';
        return 0;
}
```

Note **that std::size is only available in C++17 and later versions**. If you're using an earlier version of C++, you can use the sizeof operator to get the size of an array.

Both options will output the following:



```
The size of the array is: 5
```

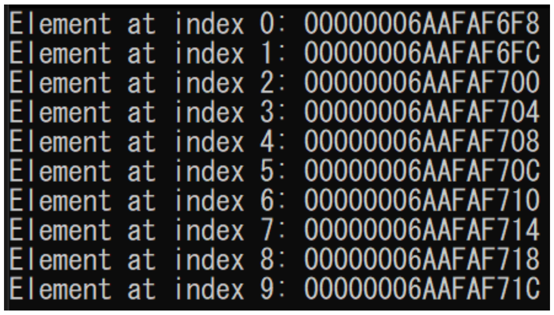
Note that size in the above examples pertain to the **number of elements, not the size of the array in bytes.**

Arrays are stored in contiguous memory locations. This means that the elements of an array are stored one after another in memory, without any gaps between them. The memory address of the first element of the array serves as the base address, and each subsequent element is located at a memory address that is offset from the base address by the size of the data type.

• An array is a collection of elements of the same data type, stored at a contiguous memory location. The array occupies *N * sizeof(T)* bytes of memory, where *N* is the number of elements in the array and *sizeof(T)* is the size of each element in bytes.

```cpp
#include <iostream>

int main() {
        int arr[]{1,2,3,4,5,6,7,8,9,10};
        int size{ std::size(arr)};
        for (int i{ 0 }; i < size; ++i) {
                std::cout << "Element at index " << i <<": " << &arr[i] << '\n';

        }
        return 0;
}
```

Output:



```
Element at index 0: 00000006AAFAF6F8
Element at index 1: 00000006AAFAF6FC
Element at index 2: 00000006AAFAF700
Element at index 3: 00000006AAFAF704
Element at index 4: 00000006AAFAF708
Element at index 5: 00000006AAFAF70C
Element at index 6: 00000006AAFAF710
Element at index 7: 00000006AAFAF714
Element at index 8: 00000006AAFAF718
Element at index 9: 00000006AAFAF71C
```

The code above prints the addresses of each element in the array. The & operator is used to get the address of a variable. The addresses are shown in **hexadecimal notation**. Each address is **4 bytes (32 bits) apart**, which corresponds to the **size of an int** element. You can observe that the addresses are contiguous because they **increase by 4 for each element**. This indicates that the elements of the array are stored in consecutive memory locations.

3. Pointers

Pointers in C++ are variables that hold memory addresses. They are used to store the location of other variables or objects in memory.

- Pointers are declared using the asterisk (*) symbol.
- Pointers are not initialized by default. It is important to initialize pointers to a known value to avoid undefined behavior.
- Pointers are often used to hold the address of another variable. The address of a variable can be obtained using the address-of operator (&).
- The dereference operator (*) is used to access the value at the address stored in a pointer. It allows you to manipulate the value that the pointer points to.
- Pointers can be declared using the asterisk (*) symbol next to the type name. For example, int* ptr; declares a pointer to an integer.

Example:

```cpp
#include <iostream>

int main() {
        int x{ 100 };
        std::cout << x << '\n'; // print the value of variable x

        int* ptr{ &x }; // ptr holds the address of x
        std::cout << *ptr << '\n';
        // use dereference operator to print the value at the address

        // that ptr is holding (which is x's address)
        return 0;
}
```

Null pointers

Null pointers are pointers that do not point to any valid memory address. They are often used to indicate the absence of a valid object or memory location. The **nullptr** keyword should be used to represent a null pointer.

Examples:

```cpp
int main() {
        int x{ 100 };
        std::cout << x << '\n'; // print the value of variable x

        int* ptr{ nullptr }; // ptr is a null pointer, and is not holding an add
ress
        ptr = &x; // ptr is now pointing at object x
        std::cout << *ptr << '\n';
        // print value of x through dereferenced ptr

        return 0;
}
```

```
#include <iostream>

int main() {
        int* ptr{ nullptr }; // ptr is a null pointer
        if (ptr == nullptr) {
                std::cout << "ptr is a null pointer\n";
        } else {
                std::cout << "ptr is not a null pointer\n";

        }

        return 0;
}
```

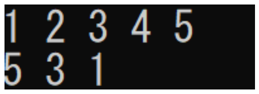
Pointer arithmetic

Pointer arithmetic refers to performing arithmetic operations on pointers in order to manipulate memory addresses. It is a fundamental concept in C++ and is often used when working with arrays or dynamically allocated memory.

- Incrementing a pointer: When you increment a pointer, it moves to the next memory location of the same type. For example, if you have a pointer to an integer (int* ptr), incrementing it (++ptr) will move it to the next integer in memory. Similarly, if you have a pointer to a character (char* ptr), incrementing it (++ptr) will move it to the next character in memory.
- Decrementing a pointer: Decrementing a pointer works in the opposite way. It moves the pointer to the previous memory location of the same type.
- Adding an offset: You can also add an offset to a pointer using the addition operator (+). This allows you to move the pointer by a specific number of elements. For example, ptr + 2 will move the pointer two elements ahead.
- Subtracting an offset: Similarly, you can subtract an offset from a pointer using the subtraction operator (-). This allows you to move the pointer back by a specific number of elements.
- Pointer comparison: Pointers can be compared using relational operators (<, >, <=, >=, ==, !=). The comparison is based on the memory addresses they point to.

Example showing pointer arithmetic:

```
#include <iostream>

int main() {
        int arr[]{ 1,2,3,4,5 };
        int* ptr{ arr }; // pointer to the first element of arr

        // Accessing array elements using pointer arithmetic
        for (int i{ 0 }; i < 5; ++i) {
                std::cout << *ptr << " ";
        // dereference the pointer to get the value
                ++ptr;
        // move the pointer to the next element
        }

        /*
                Because of the last ++ptr, the pointer now points to arr[5]
                which is out of bounds (last element is arr[4],
                so e will subtract 1 so we can point to the last element.
        */
        --ptr;
        std::cout << '\n';
        for (int i{ 0 }; i < 3; ++i) {
                std::cout << *ptr << " ";
        // dereference the pointer to get the value
                ptr -= 2;
        // move the pointer to previous 2 elements
        }
        return 0;
}
```

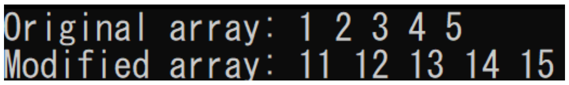
Output: 

```
1 2 3 4 5
5 3 1
```

Example of using pointer arithmetic to modify array elements

```
#include <iostream>
void modifyArray(int* arr, int size) {
        for (int i{ 0 }; i < size; ++i) {
                *(arr + i) += 10;// Equivalent to arr[i] += 10;

        // Here, *(arr + i) accesses the value at the pointer location arr + 1
        }
}
int main() {
        int arr[]{ 1,2,3,4,5 };
        int size{ std::size(arr) };


        std::cout << "Original array: ";

        for (int i{ 0 }; i < 5; ++i) {
                std::cout << arr[i]<< " ";
        // accessing  array elements using the index i

        }
        std::cout << '\n';
        modifyArray(arr, size);
        std::cout << "Modified array: ";

        for (int i{ 0 }; i < 5; ++i) {

                std::cout << arr[i] << " ";
        // accessing  array elements using the index i

        }
        std::cout << '\n';
        return 0;
}
```

Output:



```
Original array: 1 2 3 4 5
Modified array: 11 12 13 14 15
```

In the above example, we have a function modifyArray that takes a pointer to an integer (int* arr) and the size of the array. Inside the function, we use pointer arithmetic to access and modify each element of the array. The arr[i] syntax is equivalent to *(arr + i), where arr is the base address of the array and i is the index.

In the main function, we declare an array arr and calculate the size of the array using the sizeof operator. We then pass the array and its size to the modifyArray function. After calling the function, we iterate over the modified array and print its elements.

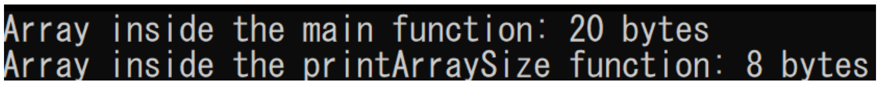
When you pass an array to a function in C++, you are actually passing a pointer to the first element of the array. This means any changes made to the array inside the function will affect the original array.

Array to pointer decay

When an array is passed as a function argument, it decays into a pointer to its first element. This means that the function receives a pointer to the array, rather than a copy of the entire array. The pointer can then be used to access and modify the elements of the array.

```
#include <iostream>
void printArraySize(int arr[]) {
        std::cout << "Array inside the printArraySize function: " <<sizeof(arr)
<< " bytes\n";
}
int main() {
        int arr[]{ 1,2,3,4,5 };
        int size{ std::size(arr) };

        std::cout << "Array inside the main function: " << sizeof(arr) << " byte
s\n";
        printArraySize(arr);
        return 0;
}
```

Output:

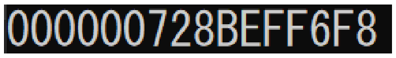


```
Array inside the main function: 20 bytes
Array inside the printArraySize function: 8 bytes
```

In the main function, we declare an array arr and calculate its size. **Int is 4 bytes** and there are 5 elements so the **size will be the number of elements multiplied by the type's size,** which results to 20. When we call the printArraySize function and pass the arr array as an argument**, the array decays into a pointer to its first element**. As a result, the sizeof(arr) expression inside the printArraySize function returns the size of the pointer, not the size of the original array.

Array decay to pointer can also be seen in the following example. This can happen when you want to access an element but forgot to use the [] operator and the index of the element

```
#include <iostream>

int main() {
        int arr[]{ 1,2,3,4,5 };

        std::cout << arr ; // arr decays to a pointer so this will print the add
ress of the first element

        return 0;
}
```

Output: 

```
000000728BEFF6F8
```