Understanding The Big O Notation

The Big O notation helps you understand how the performance of an algorithm changes as the size of the input grows. It's like a rating system that tells you how well your code will perform as you throw more and more data at it.

The "O" in Big O stands for "Order of," and it gives you an upper limit on how long an algorithm will take or how much memory it will use. The idea is to provide a "worst-case scenario" measure. It's like the speed limit sign on a road; you know you won't be going faster than that limit.

The Time complexity describes how the runtime of an algorithm grows as the size of the input increases. This is usually expressed using Big O notation, which describes an upper bound on the time complexity in the worst-case scenario.

The Space complexity describes how the amount of memory used by an algorithm grows as the size of the input increases. Just like time complexity, space complexity is also expressed using Big O notation to provide a "worst-case scenario" measure of memory usage.

How to Calculate them? What to look for in the program to calculate them?

Steps to Calculate Time Complexity:

Identify Basic Operations: Look for operations that are repeated. These could be addition, multiplication, comparisons, etc.

Count Operations: Find out how many times these basic operations are being executed. Try to express this count as a function of the input size n.

Express in Big O: Use Big O notation to describe how the number of operations grows as n increases.

Steps to Calculate Space Complexity:

Identify Memory Usage: Look for variables, data structures, and function calls that consume memory.

Count Memory: Express the total memory consumed as a function of the input size n.

Express in Big O: Use Big O notation to describe how the memory usage grows as n increases.

Here are some common Big O Notations:

O(1) - Constant time: No matter how big the data set is, the algorithm takes a constant amount of time.

O(logn) - Logarithmic time: The time taken increases logarithmically as the data set increases.

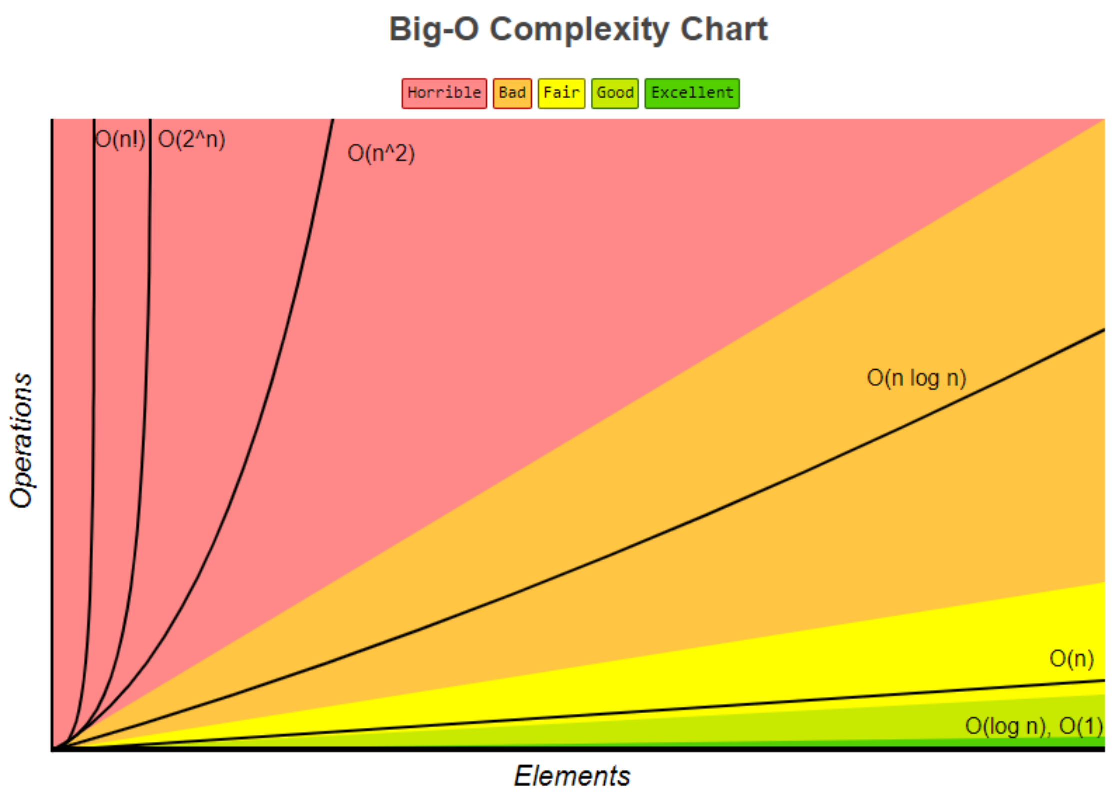
O(n) - Linear time: The time taken increases linearly with the size of the data set.

O(n^2), O(n^3), O(n^k) - Polynomial time: Time taken is a polynomial function of the data size. Time grows pretty fast as data grows. k is a constant number.

O(2^n) - Exponential Time: Time doubles with each new piece of data. Really slow.

O(n!) - Factorial Time: Extremely slow, grows faster than exponential time.

O(nlogn) - Linearithmic Time: A bit slower than linear time, but generally okay.



Img Src: https://www.bigocheatsheet.com/

O(1) – Constant Time

O(1) means that no matter how big your input is, the algorithm takes a constant amount of time to complete. The time it takes doesn't depend on the size of the data set, it's as fast as you can get.

Consider the scenario of checking your watch. Regardless of how busy you are, it takes the same quick glance to check the time on your watch. Another example would be pressing an elevator button. No matter which floor you're going to, pressing the button takes the same amount of time. Whether you must send one letter or a hundred, putting a stamp on one specific letter takes the same amount of time.

Let's understand this with a **Chai with Mrs. Sharma.**

In a small town in India, Mrs. Sharma is notorious for her... let's say, unique way of handling uninvited guests. She believes in the mantra, "The quicker you serve them, the quicker they leave."

The Usual Scenario - O(n):
In a typical household, the more the guests, the longer it takes to prepare chai. You brew, simmer, add sugar, spices, and serve. More guests mean this process takes longer.

Mrs. Sharma's Solution - O(1):
One day, a group of particularly nosy neighbors decided to drop by Mrs. Sharma's house unannounced. They wanted to see if the rumors about her husband buying a new car were true. Mrs. Sharma, ever the gracious host, decided to serve them chai. But not just any chai. She used her "special" chai powder. She whispered to her son, Raj, "Watch this. No matter if it's 1 or 10, they'll all be done with their chai at the same time." She poured hot water into cups, added her "special" chai mix, and served. True to her word, within moments, all the guests finished their chai and, feeling a sudden urge to leave, rushed out almost simultaneously.

Raj, half-amused and half-terrified, remarked, "Mom, that was wickedly efficient! Your chai serving is truly O(1)." Mrs. Sharma just winked, "That'll teach them to drop by unannounced."

Here, Mrs. Sharma's "special" chai method represents an O(1) operation. It's a constant time operation because regardless of the number of guests, her process ensures they all finish their chai simultaneously.

Examples to Understand:

Accessing an array element by its index is an O(1) operation.

```
int getElement(int arr[], int index) {
    return arr[index];
}
```

Swapping two numbers in memory takes constant time, regardless of the values of the numbers.

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

If you have a pointer to the head of a linked list, inserting a new node at the beginning is an O(1) operation.

```
typedef struct Node {
    int data;
    struct Node *next;
} Node;

void insertAtHead(Node **head, int value) {
    Node *newNode = malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = *head;
    *head = newNode;
}
```

In this example, the loop will always run 10 times, no matter what. The time it takes is constant, hence O(1).

```
void fixedLoop() {
    for (int i = 0; i < 10; i++) {
        // Some constant-time operations
        printf("This is iteration %d\n", i);
    }
}
```

Here, the outer loop runs 5 times, and the inner loop runs 4 times. The total number of iterations is 5×4=20, which is a constant. So, the time complexity is still O(1).

```
void nestedFixedLoops() {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 4; j++) {
            // Some constant-time operations
            printf("This is iteration (%d, %d)\n", i, j);
        }
    }
}
```

Even though this loop could run up to n times depending on the input array, if it contains a zero, the loop will exit early. However, this doesn't make it O(1); it's actually O(n) in the worst case when there's no zero.

```
void loopWithEarlyExit(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        // Some constant-time operations
        printf("This is iteration %d\n", i);

        if (arr[i] == 0) {
            break;
        }
    }
}
```

O(logn) Logarithmic time:

O(logn) means, the running time of an algorithm grows logarithmically as the size of the input grows. or each operation, the algorithm reduces the size of the problem by a certain factor, often halving it. This makes the algorithm very efficient, especially for large datasets.

In O(logn), you can eliminate a large portion of your problem space at each step. For example, if you're searching for a specific item in a sorted list, you don't need to check each item one by one; you can start in the middle, see if the item you're looking for is larger or smaller, and then eliminate half of the list from consideration. You can keep doing this, cutting your work in half each time, until you find what you're looking for.

Think of it like searching for a word in a dictionary. You wouldn't start from the first page and go word by word. You'd open the dictionary somewhere near the middle and adjust your search based on whether the word comes before or after the page you're currently looking at. You keep reducing the number of pages you have to look through by half each time until you find the word you're looking for.

Let us understand this with **Aman's Strategy of watching a TV serial**.

Every evening, Mrs. Kapoor's family gathers around the TV to watch the latest episode of their favorite TV serial, "Endless Desires". The show is infamous for its long-winded plots, with a single event stretching over weeks.

One day, the family's mischievous son, Aman, grew tired of waiting for the plot to progress. He figured out that for every hour of the show, only about 5 minutes were crucial – the rest was just filler, dramatic stares, and flashbacks.

The Traditional Watch - O(n):
Watching the show episode by episode, the family would spend hours, days, even weeks, waiting for a single plot point to resolve.

Aman's Strategy - O(logn):
Aman decided to implement his 'fast-forward' method. Instead of watching every episode, he'd watch one, then skip the next two, watch the next, and so on. When he felt he missed a critical plot point, he'd just rewind a bit. This way, he drastically reduced the number of episodes he had to watch. It's like binary search but for soap operas.

Soon, the entire family adopted Aman's method. Mrs. Kapoor remarked, "Beta, at this rate, we'll finish the entire 10-year series in a month!" Aman just grinned, "That's the power of logarithmic watching, Ma."

Here, Aman's strategy of skipping episodes and only backtracking when necessary, represents the logarithmic time complexity. Just as binary search reduces the problem size by half each time, Aman's method reduces the number of episodes he has to watch, but with the occasional need to revisit a skipped episode.

Examples to Understand:

```
double temp = power(x, n / 2);
if (n % 2 == 0) {
   return temp * temp;
} else {
   return x * temp * temp;
}
```

The function power is called with n/2, effectively reducing the problem size by half in each recursive call. This behavior makes it O(logn).

```
while (b != 0) {
   int temp = b;
   b = a % b;
   a = temp;
}
```

Euclid's algorithm for finding the GCD of two numbers operates in logarithmic time relative to the input numbers. It repeatedly replaces the larger number by its remainder until the numbers become zero. This gives us O(logn).

```
for (int i = 1; i <= n; i *= 2) {
   // Some constant-time operations
}
```

Here, the loop index i starts from 1 and doubles in each iteration. This means the loop will run for the values 1,2,4,8,16,… until i exceeds n. The number of times this loop runs is proportional to the number of times you can double 1 before surpassing n. This is logarithmic behavior, giving us O(logn) complexity.

```
while (n > 1) {
   // Some constant-time operations
   n = n / 2;
}
```

In this loop, the value of n is halved in each iteration. Therefore, the loop will run for as many times as n can be halved until it becomes 1 (or less). This halving behavior leads to logarithmic time complexity, O(logn).

```
for (int i = 1; i <= n; i *= 2) {
   for (int j = 1; j <= i; j++) {
      // Some constant-time operations
   }
}
```

When i = 1, the inner loop runs 1 time.
When i = 2, the inner loop runs 2 times.
When i = 4, the inner loop runs 4 times.
And so on...
If you sum up these runs, you get 1 + 2 + 4 + 8 + ... up to n. This summation is essentially a geometric series and is bounded by 2n. Therefore, despite the nested loops, the overall complexity is O(n). It's essential to understand that not every loop structure with apparent logarithmic traits (like doubling or halving) will always result in O(log n) time complexity.

O(n) Linear Time:

Imagine you have a list of numbers, and you want to check if a particular number exists in that list. The most straightforward way would be to start from the beginning of the list and check each number one by one. In the worst-case scenario, you might have to check all the numbers in the list. The time it takes to complete this task grows proportionally with the size of the list. This is a linear relationship.

For smaller datasets, a linear time algorithm might be perfectly adequate. However, as datasets grow, we might look for more efficient algorithms (like logarithmic or constant time algorithms) to handle the data more effectively.

Let's understand this with a **Dodge the Question! Scenario**.

Imagine you're at a big Indian family gathering, and you're trying to avoid that one aunty who always asks, "Beta, when are you getting married?" To spot her before she spots you, you start scanning the room, looking at each relative one by one. If she's hidden right at the back, chatting with some distant cousins, you'd have to spot every other relative before you find her. It's like playing a real-life game of "Hide and Seek", but instead of seeking friends, you're dodging matrimonial interrogations. That's the linear drama of large family gatherings.

Now, if it's just a small family dinner, evading Aunty's question is as easy as sneaking an extra ladoo when no one's looking. But imagine if it's a massive Diwali party! You might end up having to dodge questions from multiple aunties and uncles, all asking about your job, your salary, and when you'll buy a house!

Examples to Understand:

```
int sum(int arr[], int size) {
   int total = 0;
   for (int i = 0; i < size; i++) {
      total += arr[i];
   }
   return total;
}
```

In this function, the loop runs size times. The time it takes to complete grows linearly with the size of the array. The loop iterates over each element of the array once, adding its value to the total. This is a O(n) scenario.

```
int countOccurrences(int arr[], int size, int target) {
  int count = 0;
  for (int i = 0; i < size; i++) {
    if (arr[i] == target) {
      count++;
    }
  }
  return count;
}
```
The loop iterates over the array, checking if each element matches the target. If it does, the count is incremented. The loop runs size times, making the function's complexity O(n).

```
void copyArray(int source[], int destination[], int size) {
  for (int i = 0; i < size; i++) {
    destination[i] = source[i];
  }
}
```

Here, the loop iterates over each element of the source array and copies it to the destination array. The loop runs a total of size times, making the function's complexity O(n). A single loop runs based on the size of the input (size). No nested loops or recursive calls are present. The loop directly operates on each element of the array.

```
int countEven(int arr[], int size) {
  int count = 0;
  for (int i = 0; i < size; i++) {
    if (arr[i] % 2 == 0) {
      count++;
    }
  }
  return count;
}
```
This function counts the number of even integers in the array. The loop runs through each element once, checking its parity. Since the loop runs size times, the time complexity is O(n).

```
float findAverage(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return (float)sum / size;
}
```

The function calculates the average of the elements in the array. The loop iterates over the entire array once, summing up the values. The loop runs size times, giving us a time complexity of O(n).

$O(n^k)$ Polynomial Time:

Polynomial time is like those tasks that become incredibly more complicated with each extra bit of input. Think of it as a multiplier effect. If you had to complete one task, it would take some time. But if you had to complete two tasks, and then for each of those tasks, another two related tasks, the work multiplies quickly.

Let's understand this with **a WhatsApp Family group**, a place where good mornings, random health tips, and conspiracy theories abound.

Joining the Group - O(n):
Your first task is to add every family member to the group. From tech-savvy cousins to grand-aunts who think "LOL" means "Lots of Love", you've got a vast array of characters. Every new addition promises a fresh wave of chaos.

Deciphering Messages - O(n^2):
Each member, in their unique style, shares 'important' messages. For every person, you find yourself not only reading their messages but also interpreting them for older family members. "No Grandma, those 'face with tears of joy' emojis aren't because she's sad, she's laughing!" The number of clarifications multiplies with each cryptic message shared.

Damage Control - O(n^3):
Now, the real challenge. For each member and each of their messages, disputes arise. Uncle Raj thinks the moon landing was fake, while Cousin Priya is sharing a dubious remedy involving turmeric. You're constantly playing referee, trying to prevent World War III from breaking out. For every message, there's a disagreement, and for every disagreement, there's a series of follow-up discussions, apologies, and passive-aggressive comments.

By the end, managing the group feels like you've been trapped in a never-ending soap opera. And you can't even exit the group because, well, family obligations. This spiraling complexity of interpersonal dynamics, misinterpreted messages, and peacemaking efforts can be likened to the multiplying challenges of polynomial time.

Navigating the intricate web of a family WhatsApp group: it's as complex as any polynomial algorithm out there

Consider these patterns to find the Polynomial-time:

Nested Loops: Multiple nested loops, especially where each loop runs based on the input size, often indicate polynomial time complexity.
Multiple Operations on the Entire Dataset: If an algorithm performs multiple separate operations on the whole dataset, it might have polynomial complexity.

Polynomial time complexity refers to algorithms whose running time grows polynomially based on the size of the input. As the degree of the polynomial increases, the efficiency of the algorithm generally decreases, making it crucial to recognize and understand the implications of polynomial time, especially when dealing with larger datasets.

Examples to Understand:

```
for(int i = 0; i < n; i++) {
   for(int j = 0; j < n; j++) {
      // Some constant time operation
   }
}
```

For every iteration of the outer loop (which runs n times), the inner loop also runs n times. So the total operations are n * n, leading to O(n^2) time complexity.

```
for(int i = 0; i < n; i++) {
   for(int j = 0; j < n; j++) {
      for(int k = 0; k < n; k++) {
         // Some constant time operation
      }
   }
}
```

Here, for every iteration of the outermost loop, the two inner loops each run n times. This leads to a total of n * n * n operations, resulting in O(n ^3) time complexity.

```
for (int i = 0; i < n; i++) {
   // Some constant time operation
   for (int j = 0; j < n; j++) {
      // Some constant time operation
   }
}
```

Here, even though the loops are not consecutively nested, the combined operations still result in O(n ^2)) time complexity.

O(2^n) - Exponential Time:

Exponential time complexity describes an algorithm for which the time taken doubles with each additional input element. Algorithms with this time complexity often have recursive procedures that make multiple calls for smaller input sizes. They can quickly become infeasible for even moderately large input sizes due to the rapid growth in required operations.

Imagine an algorithm that for every data element you provide, it processes two possible outcomes. For the next data element, it processes two outcomes for each of the previous outcomes, so it doubles again. With each new input element, the number of operations doubles, leading to exponential growth.

Let's understand this Exponential Time with **Chintu the Chaiwala.**

Chintu runs a small tea stall in the heart of Delhi. One day, he comes up with a promotional idea to boost his business.

Starting Simple - O(1): Chintu thinks, "I'll give a free cup of chai to my regulars - Ramesh, Suresh, and Mahesh."

Engaging Friends - O(n): Excited, he shares this with his friends and says, "Share this offer with your close buddies!" Soon, a line of 20 friends forms, each wanting their free chai.

The Exponential Chaos - O(2^n): Now, here's where Chintu's plan goes haywire. Each of those 20 friends thought, "Why not share this with my group of friends?" And their friends had the same idea. So, for every friend who heard about the offer, they brought two more friends to the chai stall.

By evening, there's a massive crowd outside Chintu's small stall, all waiting for their free cup of chai. Chintu, overwhelmed, exclaims, "Arre bapu! I just wanted to give away a few cups, not drown Delhi in tea!"

This example illustrates how exponential growth can quickly spiral out of control. Just as Chintu's promotional idea led to an overwhelming crowd, algorithms with O(2 ^n) complexity can escalate operations rapidly with even a modest increase in input.

Examples to Understand:

```
int fibonacci(int n) {
   if (n <= 1)
      return n;
   return fibonacci(n - 1) + fibonacci(n - 2);
}
```

For each call to fibonacci, it branches out to two more calls (one for n−1 and another for n−2). This results in an exponential growth of calls. The time complexity is approximately O(2 ^n) for this naive recursive approach.

```
int power(int base, int exponent) {
   if (exponent == 0)
      return 1;
   if (exponent % 2 == 0)
      return power(base, exponent/2) * power(base, exponent/2);
   else
      return base * power(base, exponent/2) * power(base, exponent/2);
}
```

The function calculates the power of a number using recursion. For each recursive call when the exponent is odd, the function calls itself twice, leading to an exponential growth in the number of operations. While this isn't as efficient as possible implementations (like the "exponentiation by squaring" method), it demonstrates the O(2^n) complexity well.

```
void generateSubsets(char set[], int set_size) {
   unsigned int pow_set_size = pow(2, set_size);
   for (int counter = 0; counter < pow_set_size; counter++) {
      for (int j = 0; j < set_size; j++) {
         if (counter & (1 << j))
            printf("%c", set[j]);
      }
      printf("\n");
   }
}
```

For a set of size n, there are 2^n possible subsets. The function generates all these subsets. While the outer loop runs 2^n times, the inner loop runs n times. Hence, strictly speaking, the time complexity is O(n× 2^n), but for illustrative purposes, it gives a feel of the exponential growth.

```
void exponentialLoop(int n) {
   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= pow(2, i); j++) {
         // Some O(1) operations
         printf(".");
      }
   }
}
```

The outer loop runs n times. However, for each iteration i of the outer loop, the inner loop runs 2^i times. This causes the number of operations to grow exponentially with n.

```
void multiplicativeLoop(int n) {
   for (int i = 1; i <= n; i *= 2) {
      for (int j = 1; j <= i; j++) {
         // Some O(1) operations
         printf(".");
      }
   }
}
```

The outer loop doubles its counter with each iteration (i.e., 1, 2, 4, 8,...). The inner loop runs up to the current value of the outer loop. This structure results in an exponential growth in the number of operations.

O(n!) - Factorial Time

O(n!) (Factorial Time Complexity) is one of the highest order complexities you'll encounter, and algorithms with this complexity often become impractical even for small input sizes due to the rapid growth rate of factorial functions.

The factorial function grows very rapidly. For an input size of n, the time complexity grows at a rate of n! (n factorial). This means that the number of operations increases extremely quickly as n increases. Specifically, n! is the product of all positive integers less than or equal to n.

Factorial time complexity is most commonly associated with algorithms that generate all possible permutations of a set. As the size of the set increases, the number of permutations (or combinations, in some cases) grows factorially. It's important to recognize O(n!) complexity because it's a clear indicator that the algorithm will be highly impractical for even slightly larger input sizes. In most cases, encountering O(n!) is a signal to look for more efficient algorithms or heuristics that can solve the problem in a more reasonable time frame.

Lets understand this with a matchmaking scenario.

In a close-knit community in Delhi, Auntie Meena takes it upon herself to ensure every young person gets married. She considers herself the ultimate matchmaker.

Starting with Basics - O(1): Auntie Meena spots a potential bride, Pooja, and thinks, "Rohit from the next street would be a perfect match for her!"

Expanding the Pool - O(n): However, Auntie Meena realizes there are 10 eligible bachelors in the locality. She decides to meet each family and gather details.

The Overzealous Combination Game - O(n!): Late one evening, while sipping her masala chai, Auntie Meena gets overly ambitious. "Why not consider every possible match combination for Pooja?" For 2 bachelors, it's 2 possible match scenarios to consider. For 3, it's 6 scenarios. But for 10 bachelors, it's a mind-boggling 3,628,800 potential matchmaking scenarios! She starts envisioning dramatic scenarios like, "If Pooja marries Raj, but then there's a twist and Rohan returns from America..."

By the end of the week, Auntie Meena's living room is plastered with charts, graphs, and horoscopes. Neighbors whisper about her "matchmaking madness," and Pooja? She gets engaged her college sweetheart, rendering all of Auntie Meena's scenarios pointless.

The tale encapsulates the wild growth associated with factorial time complexity. Just as Auntie Meena's matchmaking scenarios grow out of hand, algorithms with O(n!) complexity can get unmanageable swiftly.

Examples to Understand:

```
void permute(char *str, int l, int r) {
   if (l == r)
      printf("%s\n", str);
   else {
      for (int i = l; i <= r; i++) {
         swap((str+l), (str+i));
         permute(str, l+1, r);
         swap((str+l), (str+i));  // backtrack
      }
   }
}
```

For a string of length n, the number of possible permutations is n!. The function generates all permutations of the string by recursively fixing each character in the string and permuting the remaining characters.

```
void factorialLoop(int n) {
   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= factorial(i); j++) {
         // Some O(1) operations
         printf(".");
      }
   }
}
```

The outer loop runs n times. However, for each iteration i of the outer loop, the inner loop runs i! times. This demonstrates how nested loops can contribute to factorial time complexity under certain conditions.

```
int factorial(int n) {
   int result = 1;
   for (int i = 2; i <= n; i++) {
      result *= i;
      for (int j = 1; j <= result; j++) {
         // Some O(1) operations
         printf(".");
      }
   }
   return result;
}
```

The outer loop calculates the factorial, and the inner loop iterates for each intermediate factorial value, leading to a growth that resembles factorial time complexity.

```
void printSubsets(int arr[], int n) {
   for (int i = 0; i < (1<<n); i++) {
       for (int j = 0; j < n; j++) {
          if (i & (1 << j))
             printf("%d ", arr[j]);
       }
       printf("\n");
   }
}
```

Although this is more of an exponential time complexity example (2^n), the rapid growth rate can be mistaken for factorial-like growth in certain contexts. It prints all subsets of an array.

O(nlogn) - Linearithmic Time:

Linearithmic time complexity, O(nlogn), is more efficient than linear time (O(n)) and less efficient than quadratic time O(n^2). This complexity often comes up when an algorithm divides the problem into smaller parts and processes each in linear time. Advanced sorting algorithms typically have this time complexity.

Imagine you have to sort a deck of cards. One way would be to look at each card one by one and place it in the right position. This is slow. Another approach might involve comparing each card with every other card. But there's a more efficient way: split the deck into two halves, sort each half, and then merge the two sorted halves. This divide-and-conquer method is more efficient. You break the problem down (logarithmic part) and then solve each smaller problem (linear part).

Let's understand this with a **Family Reunion**.

Planning a family reunion in India is like orchestrating a grand symphony. When Revi decides to host one, he underestimates the cascade of events that are about to unfold.

The Guest List *O*(1): Revi thinks, "I'll invite my immediate family and a few close relatives." A seemingly manageable list, right? But then, his grandmother intervenes.

Expanding the Circle -*O*(*n*): Grandma reminds him, "You can't forget your second cousin twice removed, and what about the family from the village?" The list suddenly grows from 20 to 200. Revi now has to call each one, explain the reunion, and get their RSVP.

The Seating Arrangement Nightmare - *O*(*n*log*n*): For every family that RSVPs, Revi tries to map out who gets along with whom and who had a feud 20 years ago over a misplaced goat. He spends linear time (the 'n' part) listing down all the dynamics and then logarithmic time (the 'log n' part) figuring out the perfect seating arrangement to ensure that Uncle Raj doesn't throw samosas at Uncle Vijay over some old village dispute.

As the day approaches, Revi chart of relationships and seating arrangements looks more complex than a Mumbai local train map during rush hour. He jokes, "I think it'd be easier to organize a space mission to Mars!"

In the chaos of Revi's family dynamics, the layers of complexity mirror the linearithmic nature of *O*(*n*log*n*). As the number of invitees (n) grows, the challenges of accommodating everyone's quirks and histories expand in a linearithmic pattern.

Examples to Understand:

```
for (k = l; k <= r; k++) {
   if (i > mid)
      arr[k] = aux[j++];
   else if (j > r)
      arr[k] = aux[i++];
   else if (aux[j] < aux[i])
      arr[k] = aux[j++];
   else
      arr[k] = aux[i++];
}
```

This is the merging step, where two halves of an array are merged together in sorted order. If this step is repeated log-n times (breaking and merging), the overall complexity becomes O(nlogn).

```
if (arr[i] <= arr[j])
   temp[k++] = arr[i++];
else {
   temp[k++] = arr[j++];
   count += (mid - i);  // Count inversions
}
```

Counting inversions in an array can be done using a modified merge sort. This snippet is from the merge step where inversions are counted. The process as a whole is O(nlogn) because of the divide-and-conquer nature combined with linear operations.

```
for (int i = 0; i < n; i++) {
   for (int j = 1; j < n; j *= 2) {
      // Some constant time operation
   }
}
```

The outer loop runs n times. The inner loop, however, doesn't run n times but instead runs logn times because j is doubled in each iteration. So, the combined complexity is O(nlogn).

```
for (int i = n; i > 0; i /= 2) {
   // Some constant time operation
}
for (int j = n; j > 0; j /= 2) {
   // Some constant time operation
}
```

Each loop runs for logn times due to the halving operation. But since they are not nested and run one after the other, the overall complexity remains O(nlogn).

Problems to Understand Time and Space Complexities.

Sum of all even numbers.

```
#include <stdio.h>


int sumOfEvens(int arr[], int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      if (arr[i] % 2 == 0) {
         sum += arr[i];
      }
   }
   return sum;
}

int main() {
   int arr[] = {1, 2, 3, 4, 5, 6};
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("Sum of even numbers: %d\n", sumOfEvens(arr, n));
   return 0;
}
```

Analyzing Time Complexity:

1. The function **sumOfEvens** has a single loop that iterates through the array once.
2. Inside the loop, the operation of checking if a number is even (**arr[i] % 2 == 0**) is a constant time operation $O(1)$.
3. Since the loop runs **n** times (where **n** is the size of the array), and for each iteration we are performing a constant time operation, the time complexity is $O(n)$.

Analyzing Space Complexity:

1. The only additional space we're using is the variable **sum**. No matter how large the input array is, the space used by the variable remains constant.
2. All other variables (**i** and **n**) also occupy constant space.
3. Hence, the space complexity is $O(1)$ (constant space).

Sum of Diagonal Elements in a 2-D Matrix.

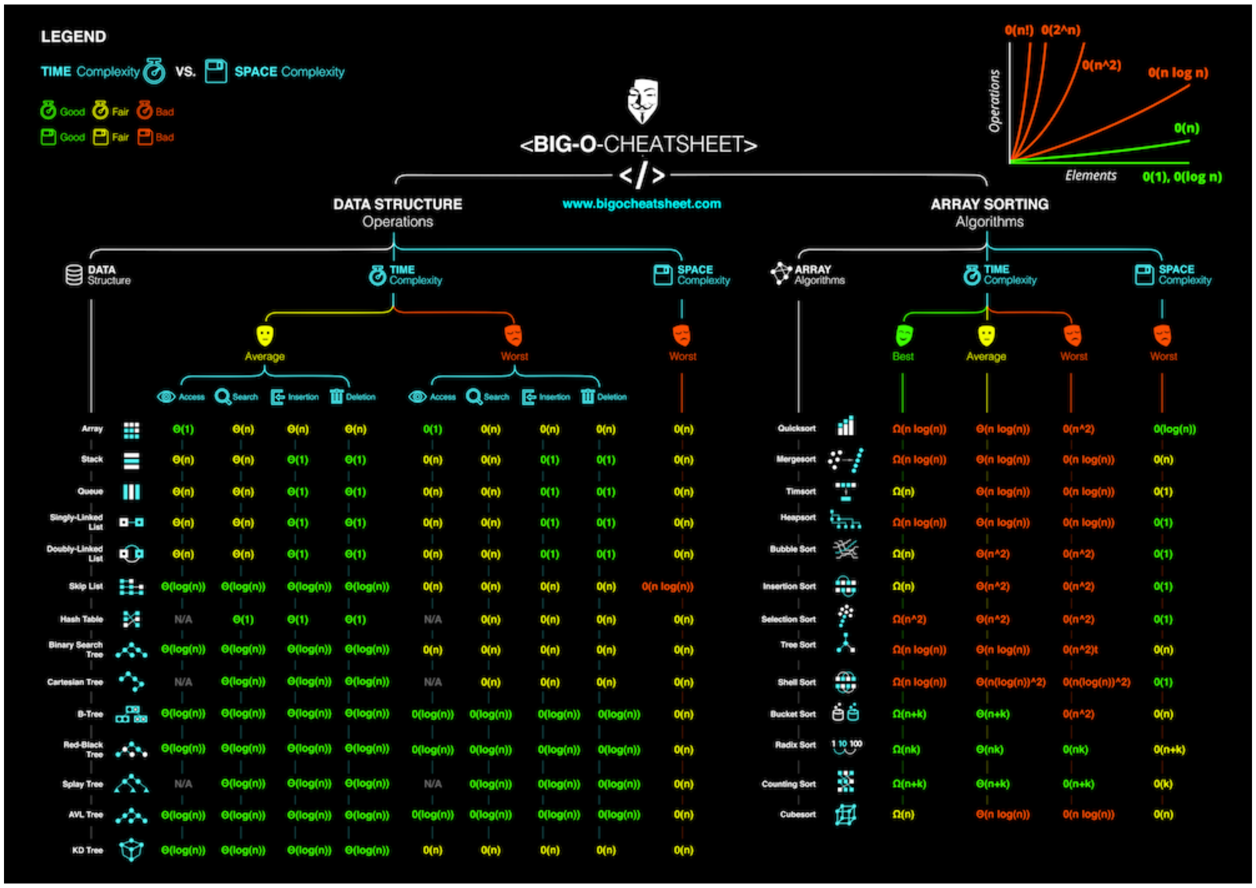
```
#include <stdio.h>

int diagonalSum(int matrix[][4], int rows, int cols) {
   int sum = 0;
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         if (i == j) {
            sum += matrix[i][j];
         }
      }
   }
   return sum;
}

int main() {
   int matrix[4][4] = {
      {1, 2, 3, 4},
      {5, 6, 7, 8},
      {9, 10, 11, 12},
      {13, 14, 15, 16}
   };
   printf("Diagonal sum: %d\n", diagonalSum(matrix, 4, 4));
   return 0;
}
```

Analyzing Time Complexity:

1. The function **diagonalSum** has two nested loops that iterate over rows and columns of the matrix.
2. The innermost operation, checking if **i == j** and then adding the matrix element, is a constant time operation ***O*(1)**.
3. The outer loop runs rows times, and the inner loop runs cols times. So, the nested loops result in rows x cols operations. Therefore, the time complexity is $\boldsymbol{O(rows \times cols)}$ or $\boldsymbol{O(n^2)}$ if you assume the matrix is of size $n \times n$.
4.

Analyzing Space Complexity:

1. The only additional space we're using is the variable sum. The size of the matrix doesn't affect how much space this variable uses.
2. All other variables (i, j, rows, cols) also occupy constant space.
3. Hence, the space complexity is $O(1)$ (constant space).



LEGEND

TIME Complexity vs. SPACE Complexity

Good Fair Bad

Good Fair Bad

<BIG-O-CHEATSHEET>

www.bigocheatsheet.com

DATA STRUCTURE Operations

DATA Structure	TIME Complexity Average Access	Average Search	Average Insertion	Average Deletion	Worst Access	Worst Search	Worst Insertion	Worst Deletion	SPACE Complexity Worst
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

ARRAY SORTING Algorithms

ARRAY Algorithms	TIME Complexity Best	Average	Worst	SPACE Complexity Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(1)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)t	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Img Src: https://www.bigocheatsheet.com/

Written By : **Yashwanth Naidu Tikkisetty**

Happy learning.

Learn together, Grow together.

Follow me to receive updates regarding Embedded Systems.

Connect with me on LinkedIn: https://www.linkedin.com/in/t-yashwanth-naidu/

T Yashwanth Naidu

https://github.com/T-Yashwanth-Naidu