Time Complexity, Searcing, and Sorting

Troy Serapio

Time Complexity

- Way to measure algorithmic runtime more objectively
- Tells us how **efficient** the algorithm is
- Not measured in milliseconds, nor by love.

Time Complexity

- Widely-used way to measure algorithmic runtime
- Tells us how **efficient** the algorithm is
- Measured in (asymptotic) **number of operations**, with respect to the **input size**.

In other words...

- As the **input size** becomes REALLY big...
- We get an **estimate** of how many operations our algorithm does
- As a **function** of the **size** of the **input**.
- Usually, we look at the **worst case scenario** of the algorithm

Example: Be There or Be Square

Given n, find n^2.

Example: Be There or Be Square

$O(n^2)$ **solution, also known as quadratic.**

```
int product = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        product = product + 1;
    }
}
```

Example: Be There or Be Square

$O(n)$ solution, also known as linear.

```
int product = 0;
for (int i = 0; i < n; i++) {
    product = product + n;
}
```

Example: Be There or Be Square

$O(1)$ solution, also known as constant.

```
int product = n * n;
```

Learn more from Reducible's video on Big-O notation!

https://www.youtube.com/watch?v=Q_1M2JaijjQ

Searching

Given a sequence of numbers A of length n, return the index of `target`.

If `target` is **not** in the array, return `-1`.

For future reference:

- A is 1-indexed
- A_i refers to the i^{th} element of A

Searching (Linear Search)

3	1	4	1	5	9	2	6	5	3	5

1. Go through every element
2. Check if element is `target`

This is done n times, thus it is $O(n)$ time complexity.

Searching (Binary Search)

1	1	2	3	3	4	5	5	5	6	9

1. Split array into two halves: **less than** **`middle`**, and **more than** **`middle`**.
   a. If `middle < target`, disregard all that is ***less*** than <u>middle</u>.
   b. If `middle > target`, disregard all that is ***more*** than <u>middle</u>.
   c. Otherwise, you found `target`!
2. Repeat Step 1 until `target` is found, or search space is empty.

Binary Search (GIF)

Respectfully yeeted from Brilliant.org, today's sponsor

Binary Search (Time Complexity)

- When we have our `middle`, we ask whether we only care about:
  - Less than `middle`
  - More than `middle`
- Thus making us **"disregard" half of our array** every split

Binary Search (Time Complexity)

- From there, we **keep halving** our array until **we can't** (1 element left)
- So the number of operations we do is "the number of times we can halve `n`?"
- Thus it is $O(\log n)$ time complexity.

Sorting

Given a sequence of numbers A of length n, return it in **non-decreasing order**.

For future reference:

- A is 1-indexed
- A_i refers to the i^{th} element of A

Sorting (Bubble Sort)

3	1	4	1	5	9	2	6	5	3	5

The following procedure is repeated `n` times:

1. Loop through each element in the sequence (A_i, where $1 \leq i < N$)
   a. If the proceeding element (A_{i+1}) is greater than A_i, swap them.
   b. Otherwise, let them be.
2. If you did not swap, that means the sequence is already sorted!

Bubble Sort (GIF)

Respectfully yeeted from Wikimedia Commons

Bubble Sort (Time Complexity)

- Since each loop through the ENTIRE array is done `n` times,
- we perform roughly n^2 operations at worst.
- Therefore, the time complexity is $O(n^2)$.

Sorting (Insertion Sort)

3	1	4	1	5	9	2	6	5	3	5

For each element A_i (for $1 \leq i \leq n$):

1. Compare A_i to **all elements to its left**
2. **Insert** it such that the **left part of A is still sorted**

Insertion Sort (GIF)

Respectfully yeeted from Wikimedia Commons

Insertion Sort (Time Complexity)

- For each element of A, we perform `n` operations (in the worst case) to find the **rightful place for `element`**.
- We also have n elements in A.
- Thus, in total, we perform roughly n^2 operations.
- Consequently, it has a time complexity of $O(n^2)$.

Got more questions and clarifications?

Ask the Reboot Discord server!!