TIME AND SPACE COMPLEXITY EXERCISES

In this series of exercises, we'll get you up and running with time and space complexity.

In the guide, we will quickly highlight the key concepts that you need to know for your interview. If you're unfamiliar with any of these, I highly recommend taking some time to review them.

Then, I've provided you with a series of exercises to help you get up and running quickly. I recommend setting aside 2-3 days to work through these exercises. Simply start from the top and work through them as much as you have time.

Make sure to track the time you spend here in your Interview Prep Tracker. You can categorize this as "Study" time.

TABLE OF CONTENTS

TIME AND SPACE COMPLEXITY EXERCISES

Exercise Set #1 - Asymptotic Notation

1. In the Big Oh Notation lesson, we learned that `f(x) == O(g(x))` if and only if `M` and x_0 exist such that `Mg(x) >= f(x)` for all $x > x_0$.

   For each of the following, identify 3 possible values for `M` and x_0 that demonstrate that `f(x) == O(g(x))`.

   If you are unsure, I recommend using a tool like Wolfram Alpha or Google Sheets to graph each function for a visual representation.

   a. `f(x) = x/2 + 25`
      `g(x) = x`

      x0=2 and M=13 => Mg(x0) >= f(x0) => 13*2 >= 2/2+25 => 26 >= 26
      x0=4 and M=7 => Mg(x0) >= f(x0) => 7*4 >= 4/2+25 => 28 >= 27
      x0=6 and M=5 => Mg(x0) >= f(x0) => 5*6 >= 6/2+25 => 30 >= 28

   b. `f(x) = (x-1)(x-2)`
      $g(x) = x^2$

      x0=1 and M=1 => Mg(x0) >= f(x0) => 1*(1^2) >= (1-1)(1-2) => 1 >= 0
      x0=2 and M=1 => Mg(x0) >= f(x0) => 1*(2^2) >= (2-1)(2-2) => 4 >= 0
      x0=4 and M=1 => Mg(x0) >= f(x0) => 1*(4^2) >= (4-1)(4-2) => 16 >= 6

   c. `f(x) = x+100`
      $g(x) = x * \log_2(x)$

      x0=2 and M=51 => Mg(x0) >= f(x0) => 51*(2*log2(2)) >= (2+100) => 102 >= 102
      x0=4 and M=13 => Mg(x0) >= f(x0) => 13*(4*log2(4)) >= (4+100) => 104 >= 104
      x0=6 and M=6 => Mg(x0) >= f(x0) => 6*(6*log2(6)) >= (6+100) => 108 >= 106

   d. `f(x) = x * (x + 1) / 2`
      $g(x) = x^2$

      x0=1 and M=1 => Mg(x0) >= f(x0) => 1*(1*((1+1)/2)) >= 1 >= 1
      x0=3 and M=2 => Mg(x0) >= f(x0) => 2*(3*((3+1)/2)) >= 12 >= 9
      x0=5 and M=2 => Mg(x0) >= f(x0) => 2*(5*((5+1)/2)) >= 30 >= 25

2. Convert each of the following functions into it's asymptotic (big oh) notation.

   a. `f(x) = (2x+5)/2` — O(n)

   b. `f(x) = (x-1)(x-2)` — O(n^2)

   c. $f(x) = \log(x) * (x^2 + x - 2)$ — O(n^2 * log(n))

   d. $f(x) = (x + 1)^2$ — O(n^2)

   e. $f(x) = x^2 + 2^x + x*\log(x)$ — O(2^n)

Exercise Set #2 - Computing Toy Time Complexities

For each of the following code samples, compute the time complexity.

Remember the two strategies that you can use when in doubt:
1. Expand the function manually
2. Create a summation and plug it into Wolfram Alpha

1.
```
void myFunc(int x) {
    for (int i = 0; i < 100; i++) {
        print(x+i);
    }
}
```
O(n)

2.
```
void myFunc(int[] arr) {
    for (int i = 0; i < 100; i++) {
        arr.sort();
        arr.randomize(); // Randomize order of elements, O(nlogn) time
    }
}
```
O(n^2*log(n))

3.
```
void myFunc(int x) {
    for (int i = 0; i < x; i++) {
        for (int j = i+2; j < x; j++) {
            print(i+j);
        }
    }
}
```
O(n*n) = O(n^2)

4.
```
void myFunc(int x) {
    while (x > 0) {
        print(x % 2);
        x = x/2;
    }
}
```
O(log_2 n)

Exercise Set #3 - Computing Real-World Time Complexities

For each of the following code samples, compute the time complexity.

Remember the two strategies that you can use when in doubt:
1. Expand the function manually
2. Create a summation and plug it into Wolfram Alpha

1.
```
List<int[]> myFunc(int[] arr, int target) {
    List<Integer[]> result = new ArrayList<>();
    int i = 0;
    int j = arr.length-1;

    while (i < j) {
        int sum = arr[i] + arr[j];
        if (sum == target) {
            result.add(new Integer[]{arr[i], arr[j]});
            while (arr[i] == arr[i+1]) i++;
            i++;
        }

        if (sum < target) i++;
        if (sum > target) j--;
    }
}
```

O(arr.length)

2.
```
int myFunc(int[] coins, int amount) {
    if (amount < 0) return -1;
    int[] dp = new int[amount+1];

    for (int i = 1; i < dp.length; i++) {
        int minCoins = MAX_VALUE;
        boolean foundValid = false;
        for (int c : coins) {
            if (i - c >= 0 && dp[i-c] >= 0) {
                minCoins = min(minCoins, dp[i-c]+1);
                foundValid = true;
            }
        }
        if (foundValid) dp[i] = minCoins;
        else dp[i] = -1;
    }
    return dp[amount];
}
```

O(amount * coins.length)

3.
```
int[] myFunc(int[] nums) {
    int length = nums.length;
    int[] L = new int[length];
    int[] R = new int[length];

    int[] answer = new int[length];
    L[0] = 1;
    for (int i = 1; i < length; i++) {
        L[i] = nums[i - 1] * L[i - 1];
    }

    R[length - 1] = 1;
    for (int i = length - 2; i >= 0; i--) {
        R[i] = nums[i + 1] * R[i + 1];
    }

    for (int i = 0; i < length; i++) {
        answer[i] = L[i] * R[i];
    }

    return answer;
}
```

O(nums.length)

4.
```
List<int[]> myFunc(int[][] intervals) {
    if (intervals.length <= 1)
        return intervals;

    intervals.sort();
    List<int[]> result = new ArrayList<>();
    int[] newInterval = intervals[0];
    result.add(newInterval);

    for (int[] interval : intervals) {
        if (interval[0] <= newInterval[1]) {
            newInterval[1] = max(newInterval[1], interval[1]);
        } else {
            newInterval = interval;
            result.add(newInterval);
        }
    }
    return result.toArray(new int[result.size()][]);
}
```

O(intervals.length*log(intervals.length))

Exercise Set #4 - Computing Toy Space Complexities

For each of the following code samples, compute the space complexity.

Remember from the lesson the differences between space and time complexity:
1. We only consider EXTRA space. Space used for the input and output do not count towards the space complexity
2. Space can be reused whereas time cannot

1.
```
int[] myFunc(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        int temp = arr[i];
        arr[i] = arr[arr.length-i-1];
        arr[arr.length-i-1] = temp;
    }
    return arr;
}
```
O(1)

2.
```
boolean myFunc(String s1, String s2) {
    s2 = s2.reverse();
    return s1 == s2;
}
```
O(s2.length)

3.
```
ListNode myFunc(ListNode head) {
    ListNode prev = null;
    ListNode curr = head;

    while (curr != null) {
        ListNode nextTemp = curr.next;
        curr.next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}
```
O(1)

4.
```
void myFunc(String s) {
    int[] count = new int[256];
    for (int i = 0; i < s.length(); i++) count[s.charAt(i)]++;
}
```
O(1)

Exercise Set #5 - Computing Real-World Space Complexities

For each of the following code samples, compute the space complexity.

Remember from the lesson the differences between space and time complexity:
1. We only consider EXTRA space. Space used for the input and output do not count towards the space complexity
2. Space can be reused whereas time cannot

1.
```
boolean myFunc(String s1, String s2) {
    if (s1.length() != s2.length()) return false;
     Map<Character, Integer> chars = new HashMap<>();

     for (char c : s1.toCharArray()) {
         int count = 1;
         if (chars.containsKey(c)) {
             count = chars.get(c)+1;
         }
         chars.put(c, count);
     }

     for (char c : s2.toCharArray()) {
         if (!chars.containsKey(c)) return false;
         int count = chars.get(c);
         if (count == 0) return false;
         chars.put(c, count-1);
     }
    return true;
}
```

O(s1.length)

2.
```
int myFunc(int[] coins, int amount) {
    if (amount < 0) return -1;
    int[] dp = new int[amount+1];

    for (int i = 1; i < dp.length; i++) {
        int minCoins = MAX_VALUE;
        boolean foundValid = false;
        for (int c : coins) {
            if (i - c >= 0 && dp[i-c] >= 0) {
                minCoins = min(minCoins, dp[i-c]+1);
                foundValid = true;
            }
        }
        if (foundValid) dp[i] = minCoins;
        else dp[i] = -1;
    }
    return dp[amount];
}
```

O(amount)

3.
```
int[] myFunc(int[] nums) {
    int length = nums.length;
    int[] L = new int[length];
    int[] R = new int[length];

    int[] answer = new int[length];
    L[0] = 1;
    for (int i = 1; i < length; i++) {
        L[i] = nums[i - 1] * L[i - 1];
    }

    R[length - 1] = 1;
    for (int i = length - 2; i >= 0; i--) {
        R[i] = nums[i + 1] * R[i + 1];
    }

    for (int i = 0; i < length; i++) {
        answer[i] = L[i] * R[i];
    }

    return answer;
}
```

O(nums.length)

4.
```
List<int[]> myFunc(int[][] intervals) {
    if (intervals.length <= 1)
        return intervals;

    intervals.sort();
    List<int[]> result = new ArrayList<>();
    int[] newInterval = intervals[0];
    result.add(newInterval);

    for (int[] interval : intervals) {
        if (interval[0] <= newInterval[1]) {
            newInterval[1] = max(newInterval[1], interval[1]);
        } else {
            newInterval = interval;
            result.add(newInterval);
        }
    }
    return result;
}
```

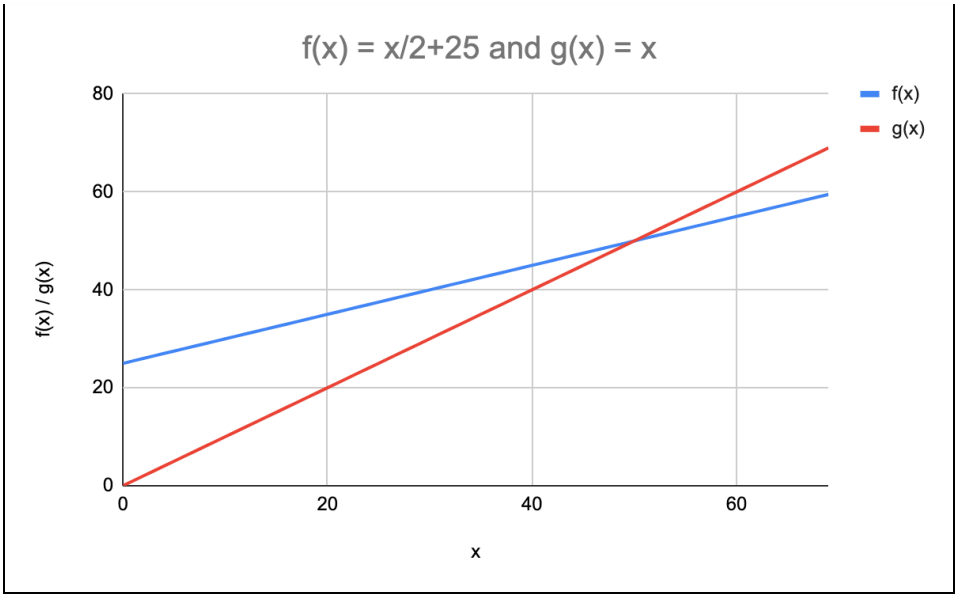
O(1)

TIME AND SPACE COMPLEXITY SOLUTIONS

Exercise Set #1 Solutions

1. For each of these problems, there are infinite possibiliites for `M` and x_0. As long as `f(`x_0`)` <= `M*g(`x_0`)`, your answer is correct. Below are simply examples of what `M` and x_0 *could* be.

a.



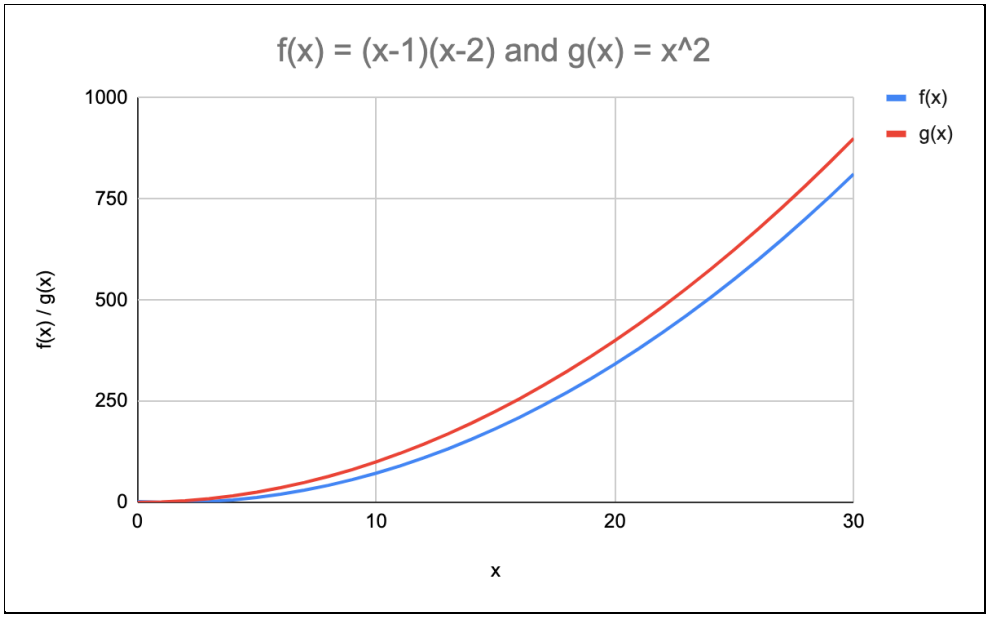
Experiment on Wolfram Alpha

`M = 1` and x_0 `= 50`
`f(50) = 50/2 + 25 = 50`
`1*g(50) = 50`

`M = 5` and x_0 `= 20`
`f(20) = 20/2 + 25 = 35`
`5*g(20) = 5*20 = 100`

`M = 10` and x_0 `= 10`
`f(10) = 10/2 + 25 = 30`
`10*g(10) = 10*10 = 100`

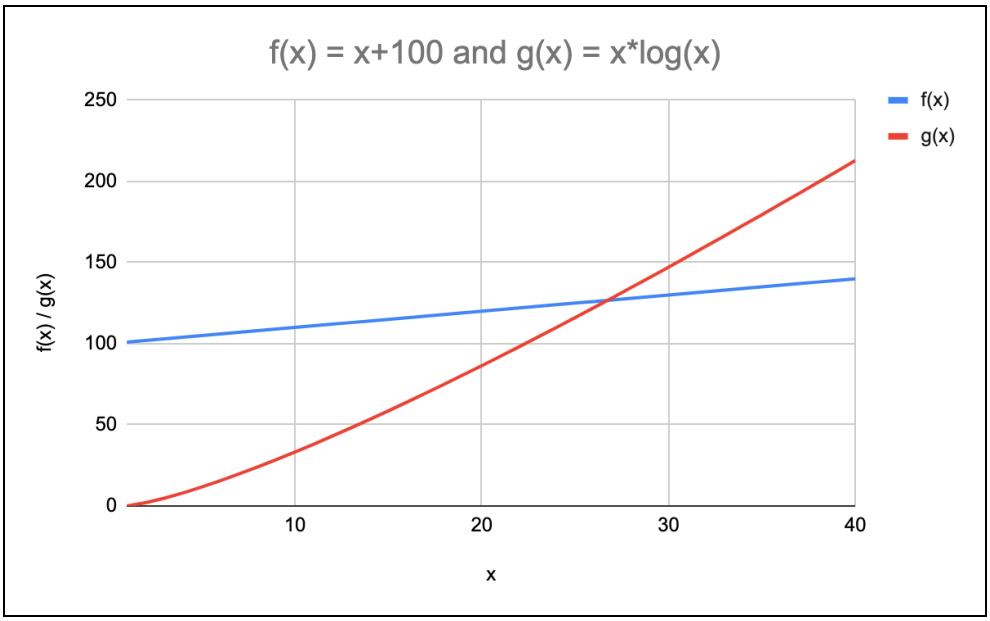
b.



Experiment on Wolfram Alpha

M = 1 and x_0 = 3
f(3) = (3-1)(3-2) = 2*1 = 2
1*g(3) = 3^2 = 9

c.

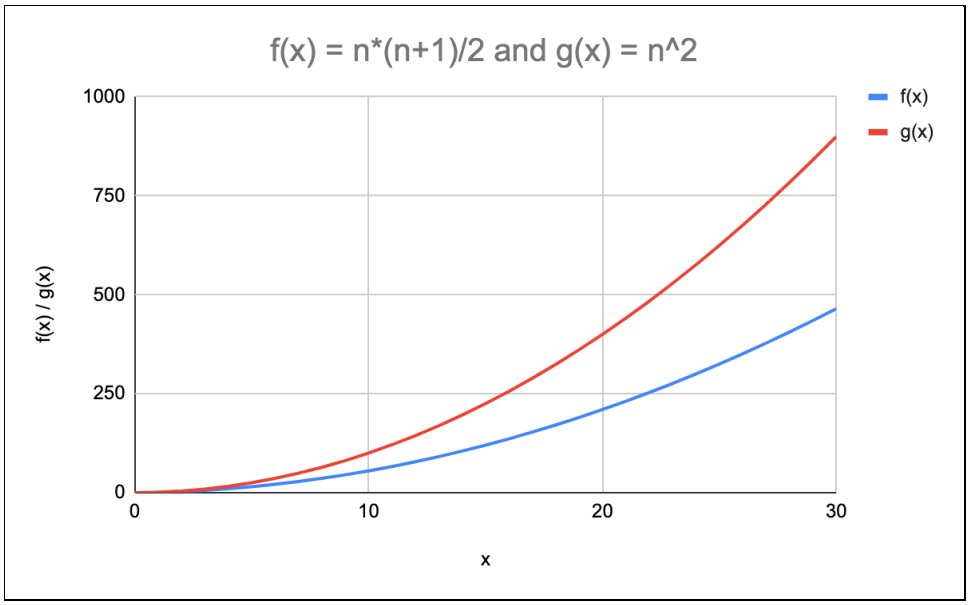


Experiment on Wolfram Alpha

M = 1 and x_0 = 28
f(28) = 28+100 = 128
1*g(x) = 28 * $\log_2(28)$ = 134.6

M = 5 and x_0 = 10
f(10) = 10+100 = 110
5*g(10) = 5*10*$\log_2(10)$ = 166.1

M = 60 and x_0 = 2
f(2) = 2+100 = 102
60*g(2) = 60*2*$\log_2(2)$ = 120

d. 

[Experiment on Wolfram Alpha]

M = 1 and $x_0 = 5$
f(5) = 5*(5+1)/2 = 15
1*g(x) = 5^2 = 25

2. To find the asymptotic notation, we simply extract the highest-order term from each function.

a. $O(x)$
b. $O(x^2)$ - If we expand this function we get $x^2 - 3x + 2$
c. $O(x^2 log(x))$ - If we expand this function we get $x^2 log(x) + x log(x) - 2 log(x)$
d. $O(x^2)$ - If we expand this function we get $x^2 + 2x + 1$
e. $O(2^x)$ - This is by far larger than any of the other terms

Exercise Set #2 Solutions

1. `O(1)`
   This is an example where we have a for loop but the number of times we loop through it does not change regardless of the size of the input. The input has no impact on how fast this code will run and therefore it is `O(1)` time.

2. `O(n log n)`
   Like with the last problem, we have a loop that is not related to the size of the input. Therefore, our time is only controlled by what is inside the loop. In this case, we have `sort()` and `randomize()`, both of which take `O(n log n)` time (for standard functions like sorting, just assume that the runtime is standard). We're doing each 100 times, but that's fixed, so it functions the same as a coefficient that we remove. Therefore we basically get `f(x) = 100*((n log n) + (n log n))`, which simplifies to `O(n log n)`.

3. $O(x^2)$
   This is a good opportunity to either manually expand our code to see what we get or plug it into Wolfram Alpha. Let's look at both options.

   Manual Expansion:
   Write out the amount of work for each iteration of the inner loop from `i = 0` to `x`. In this case we go from `j = i+2` to `x`, so the amount of work is `x - (i + 2)`:

   ```
   (x-(0+2)) + (x-(1+2)) + (x-(2+2)) + ... + (x-(x-1+2))
   ```

   Simplifying a bit, we get:

   ```
   (x-2) + (x-3) + (x-4) + ... + (x-(x+1))
   ```

   Now our `for` loop will not run if `i >= x`, so we can truncate the last few values and get:

   ```
   (x-2) + (x-3) + (x-4) + ... + (x-x)
   ```

   With that, if we continue to rewrite our function we ultimately get something that looks like this:

   x^2 `- ...`

   That means, our highest-order term in our function is x^2 and therefore we get that our complexity is $O(x^2)$.

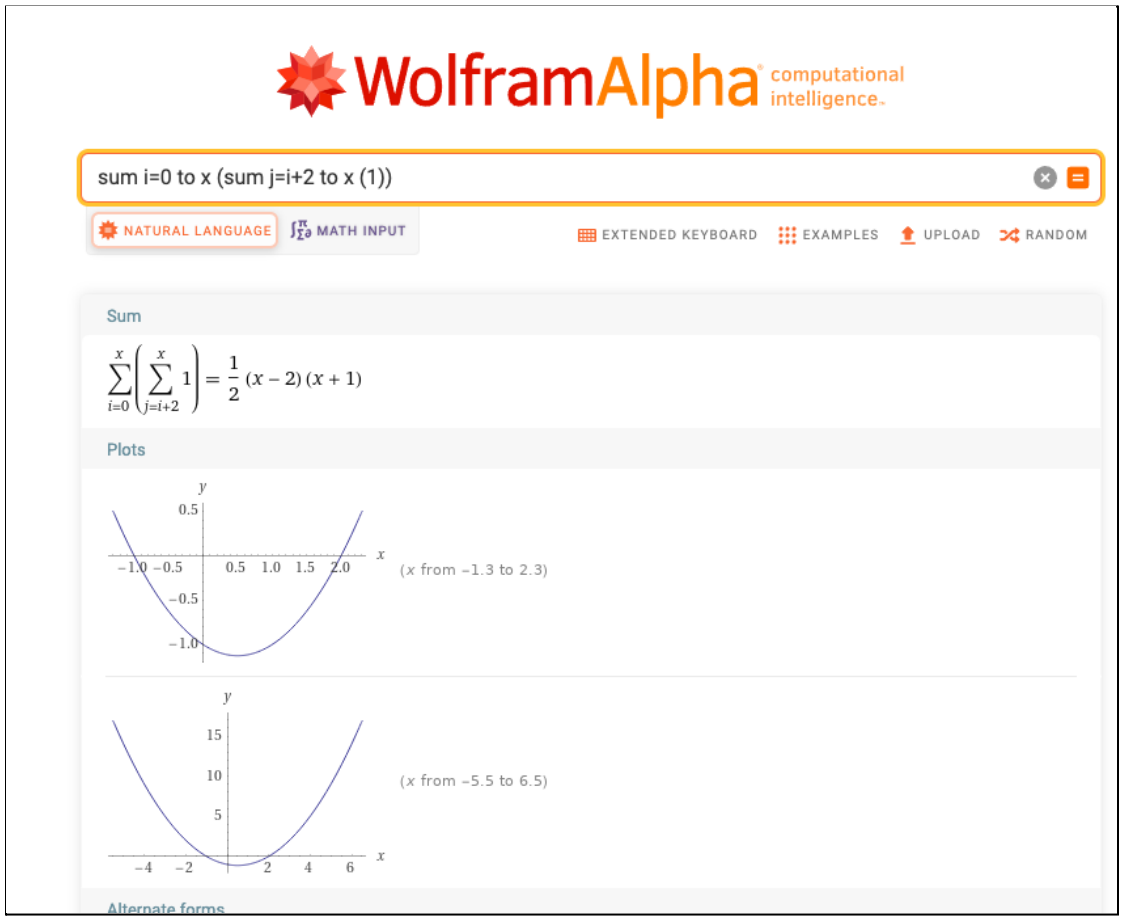
Wolfram Alpha:
Doing this by computing a sum in Wolfram Alpha is even easier because we don't need to do all the expansion, we just need to define our sums, which are really just our for loops.

Our outer for loop becomes ∑ `i=0 to x` (inner for loop).

The inner for loop becomes ∑ `j=i+2 to x (1)`. The amount of work inside the loop is constant so we just put 1.

Combined together we get ∑ `i=0 to x (∑ j=i+2 to x (1))`. We can type this into Wolfram Alpha as "sum i=0 to x (sum j=i+2 to x (1))"

Here is what the output looks like:



[Check it out for yourself here.]

The very top line of the results gives us the full result of our function. Expanding that, we get $O(x^2)$.

4. `O(log x)`
   By now you should be starting to recognize this pattern. Whenever we divide the size of our input on each step, that is likely logarithmic time.

Exercise Set #3 Solutions

1. `O(arr.length)`
   Each iteration through the while loop, we either increment `i` or decrement `j` at least once. Our stopping condition is when `i >= j`. Since `i` starts at `0` and `j` starts at `arr.length-1`, we will traverse the whole array once with the combination of `i` and `j`.

2. `O(amount * coins.length)`
   Our outer for loop iterates from `1` to `amount`. During each loop, we iterate over `coins`.

3. `O(nums.length)`
   We iterate over `nums.length` multiple times in sequence, so our complexity is just `O(nums.length)`.

4. `O(intervals.length * log(intervals.length))`
   We sort the intervals array, and everything else only takes linear time.

Exercise Set #4 Solutions

1. `O(1)`
   While we are rearranging values inside of the array, the array is not allocated inside this function so it doesn't count towards the space complexity.

2. `O(s2.length)`
   In many languages strings are immutable. Therefore, reversing s2 actually creates a copy of s2 with the characters reversed.

3. `O(1)`
   We are manipulating the nodes of a list that was allocated outside of our current function. The only memory allocation we do is creating a single temp node at a time, which is constant space.

4. `O(1)`
   Although we are allocating an array, it is fixed-size. Since it is not dependent on the size of the input it is still `O(1)` space complexity.

Exercise Set #5 Solutions

1. `O(s1.length)`
   The size of our HashMap is dependent on the number of characters in `s1`. That is the only space we're allocating.

   NOTE: If we know that `s1` is only ASCII characters, then the same principle for 4.4 applies and the complexity is `O(1)` because there are a max number of unique characters and therefore a fixed max size of our HashMap

2. `O(amount)`
   We are allocating the dp array of size `amount+1`.

3. `O(nums.length)`
   We allocate `L` and `R`, both of size `nums.length`.

4. `O(1)`
   While we are allocating a list here, that list is required to contain the output and therefore we do not count it because it is not EXTRA space.