

DAA- HANDS ON3

NAVYA SREE CHAGAMREDDY

```
function x = f(n)
    x = 1;
    for i = 1:n
        for j = 1:n
            x = x + 1;
```

1. Find the runtime of the algorithm mathematically (I should see summations).

DAA
Hands-On3

Name: Navya Sree
Chagamreddy
ID: 1002197805

Function 1:

```
function x = f(n)
    x = 1;
    for i = 1:n
        for j = 1:n
            x = x + 1;
```

- 1) Find the runtime of the algorithm mathematically.

→ The above algorithm has 2 nested loops;

The outer loop runs from $i = 1$ to $i = n$,

The inner loop runs from $j = 1$ to $j = n$,

So in each inner loop iteration, the variable 'x' is incremented by 1

→ Now, total no. of iterations is;

$$T(n) = 1 + \sum_{i=1}^n \sum_{j=1}^n 1$$

The inner summation $\sum_{j=1}^n 1$ says the no. of inner loop iterations i.e., n since the inner loop runs n times for each outer loop iteration;

Substituting in outer summation,

$$T(n) = 1 + \sum_{i=1}^n n = 1 + n \cdot \sum_{i=1}^n 1$$

$\sum_{i=1}^n 1 = n$, as there are 'n' terms in the sum

∴ Runtime for the algorithm is $O(n^2)$

2. Time this function for various n e.g. $n = 1, 2, 3, \dots$. You should have small values of n all the way up to large values. Plot "time" vs " n " (time on y-axis and n on x-axis). Also, fit a curve to your data, hint it's a polynomial.

Code:

```
import matplotlib.pyplot as plt

def runtime(n):

    return n**2

values = list(range(1, 201))

runtime = [runtime(n) for n in values]

# Plotting

plt.plot(values, runtime, marker='o', linestyle='-', color='b')

plt.xlabel('input size')

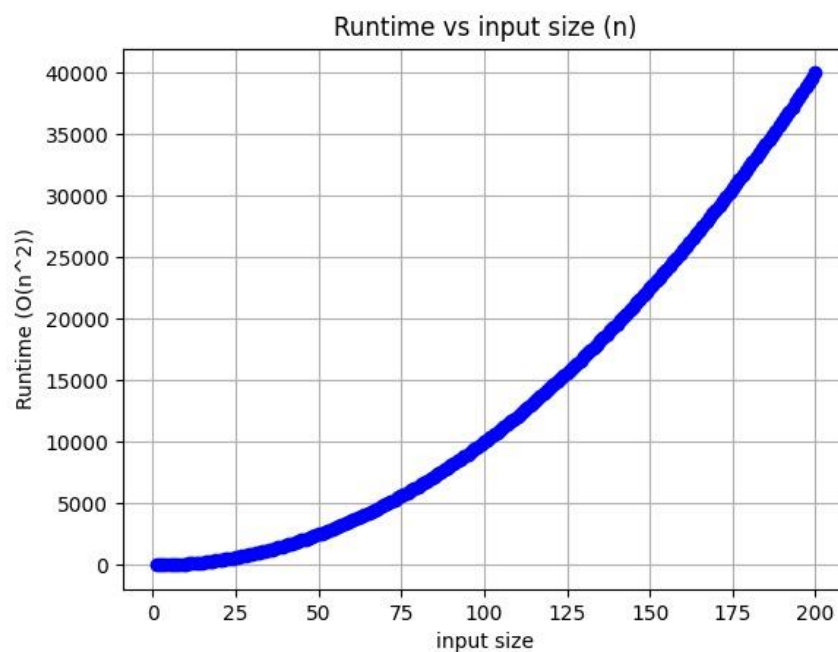
plt.ylabel('Runtime (O(n^2))')

plt.title('Runtime vs input size (n)')

plt.grid(True)

plt.show()
```

Plotting Graph:



3. Find polynomials that are upper and lower bounds on your curve from #2. From this specify a big-O, a big-Omega, and what big-theta is.

3) Given function, $T(n) = n^2 + 1$

Considering the upper bound $U(n) = 2n^2$
Lower bound $L(n) = 0.8n^2$

1) Big O (O):

$T(n) = n^2 + 1$ is $O(n^2)$

because there exists the constants C & n_0
i.e, $T(n) \leq C \cdot n^2 \forall n \geq n_0$

2) Big-Omega (Ω):

$T(n) = n^2 + 1$ is $\Omega(n^2)$

because there exists the constants C & n_0
i.e, $T(n) \geq C \cdot n^2 \forall n \geq n_0$

3) Big-Theta (Θ):

$T(n) = n^2 + 1$ is $\Theta(n^2)$

because there exists the constants C_1, C_2, n_1, n_2
i.e, $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2 \forall n \geq \max(n_1, n_2)$

Code:

```
import matplotlib.pyplot as plt

def quadratic_function(n):

    return n**2

input_values = list(range(1, 500))
```

```

runtime = [quadratic_function(n) for n in input_values]

plt.plot(input_values, runtime, label='O(n^2)', color='blue')

upper_bound = [2 * n ** 2 for n in input_values]

lower_bound = [0.8 * n ** 2 for n in input_values]

plt.plot(input_values, upper_bound, label='Upper Bound', linestyle='--', color='orange')

plt.plot(input_values, lower_bound, label='Lower Bound', linestyle='--', color='green')

plt.xlabel('Input Size (n)')

plt.ylabel('Runtime')

plt.title('Runtime V/S Input Size (n)')

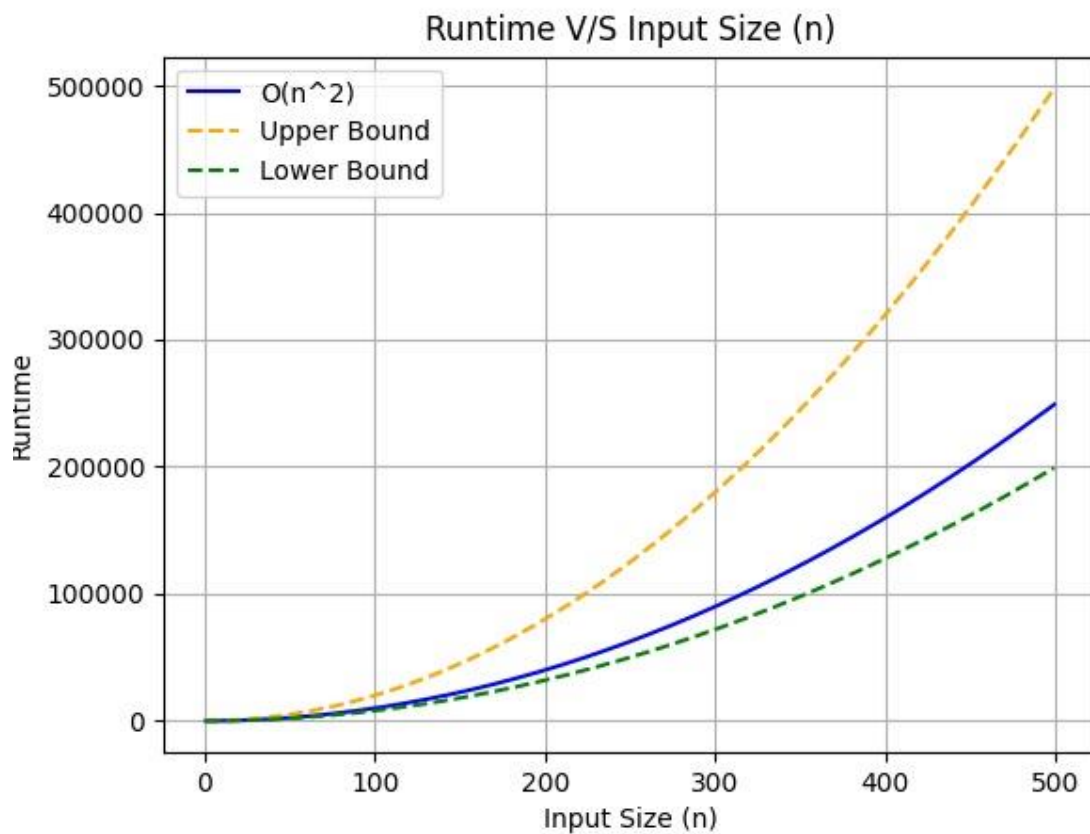
plt.legend()

plt.grid(True)

plt.show()

```

Graph:



4. Find the approximate (eye ball it) location of "n_0" . Do this by zooming in on your plot and indicating on the plot where n_0 is and why you picked this value. Hint: I should see data that does not follow the trend of the polynomial you determined in #2.

Code:

```
import matplotlib.pyplot as plt

def quadratic_function(n):

    return n**2

input_values = list(range(1, 200))

runtime = [quadratic_function(n) for n in input_values]

plt.plot(input_values, runtime, label='O(n^2)', color='blue')

upper_bound = [2 * n ** 2 for n in input_values]

lower_bound = [0.8 * n ** 2 for n in input_values]

plt.plot(input_values, upper_bound, label='Upper Bound', linestyle='--', color='orange')

plt.plot(input_values, lower_bound, label='Lower Bound', linestyle='--', color='green')

plt.xlabel('Input Size (n)')

plt.ylabel('Runtime')

plt.title('Input Size (n) V/S Runtime')

plt.legend()

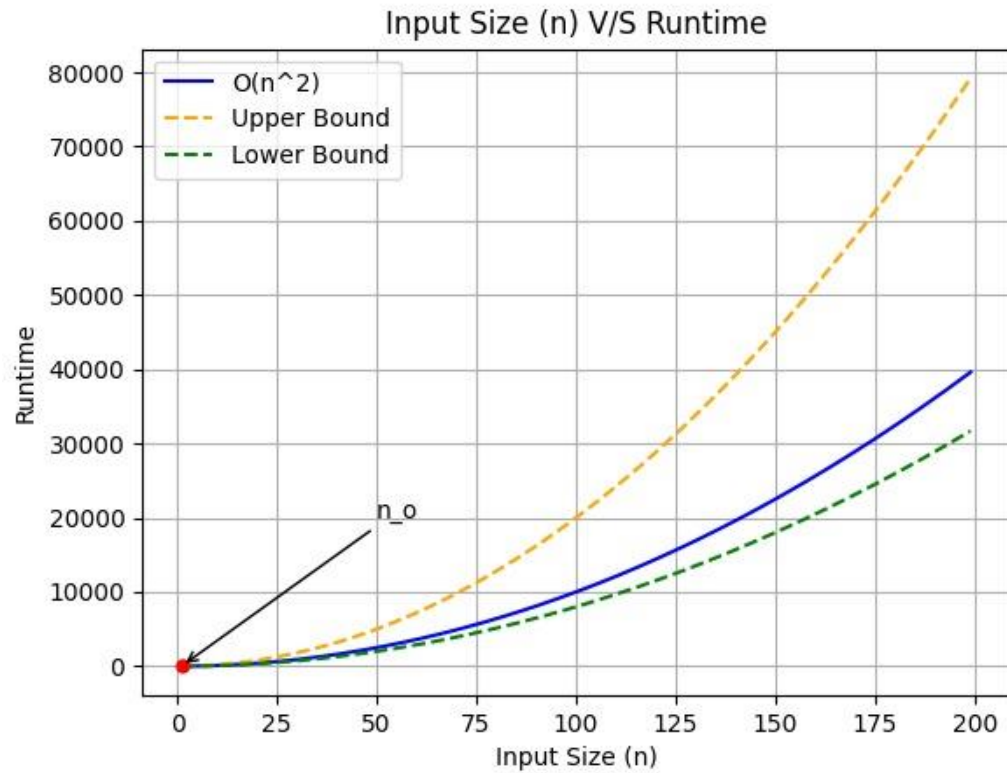
plt.plot(1, 1, marker='o', markersize=5, color='red')

plt.annotate('n_o', xy=(1, 1), xytext=(50, 20000),arrowprops=dict(facecolor='black', arrowstyle='->'))

plt.grid(True)

plt.show()
```

Graph:



Reason to pick $n_0 = 1$:

I have selected $n_0=1$ because at this particular input size, a significant change in the runtime behaviour is apparent. Beyond this threshold, the deviation from the expected polynomial trend becomes more pronounced, indicating that the quadratic term starts to exert a more noticeable deviation on the runtime.

If I modified the function to be:

```
x = f(n)
x = 1;
y = 1;
for i = 1:n
    for j = 1:n
        x = x + 1;
        y = i + j;
```

4. Will this increase how long it takes the algorithm to run (e.x. you are timing the function like in #2)?

Function 2 :

The modified function to be:

```
x = f(n)
x = 1;
y = 1;
for i = 1:n
    for j = 1:n
        x = x + 1;
        y = i + j;
```

4)

Ans:

In this function 2, there are 2 nested loops ranging from 1 to 'n'. However, there are additional constant time operations inside the inner loop i.e. ($y = i + j$). The dominant factor is still the nested loops, so, the time complexity remains $O(n^2)$.

And yes, the addition of an extra operation within the nested loop results in increasing the overall runtime.

5. Will it effect your results from #1?

5) No, the runtime for the function #2 is still $O(n^2)$

$$= c + \sum_{i=1}^n \sum_{j=1}^n 1$$

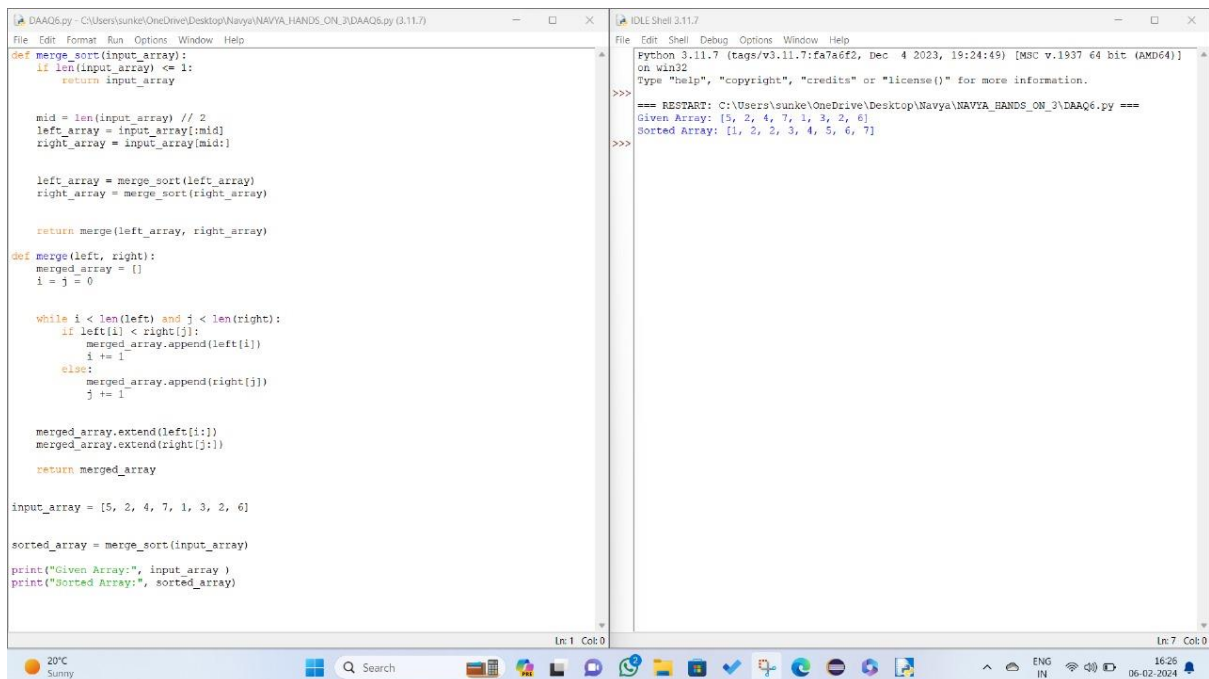
$$\text{work} \sum_{k=1}^n 1 = n$$

$$= c + n(n)$$

$$= c + n^2$$

$$= \underline{\underline{O(n^2)}}$$

6. Implement merge sort, upload your code to github and show/test it on the array [5,2,4,7,1,3,2,6].



The image shows a screenshot of a Python IDE with two windows. The left window, titled 'DAAQ6.py', contains the implementation of the merge sort algorithm. The right window, titled 'IDLE Shell 3.11.7', shows the execution output.

```
def merge_sort(input_array):
    if len(input_array) <= 1:
        return input_array

    mid = len(input_array) // 2
    left_array = input_array[:mid]
    right_array = input_array[mid:]

    left_array = merge_sort(left_array)
    right_array = merge_sort(right_array)

    return merge(left_array, right_array)

def merge(left, right):
    merged_array = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged_array.append(left[i])
            i += 1
        else:
            merged_array.append(right[j])
            j += 1

    merged_array.extend(left[i:])
    merged_array.extend(right[j:])

    return merged_array

input_array = [5, 2, 4, 7, 1, 3, 2, 6]

sorted_array = merge_sort(input_array)

print("Given Array:", input_array)
print("Sorted Array:", sorted_array)
```

The output in the shell window is as follows:

```
Python 3.11.7 (tags/v3.11.7:fa7a6f2, Dec 4 2023, 19:24:49) [MSC v.1937 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:\Users\sunke\OneDrive\Desktop\Navya\NAVYA_HANDS_ON_3\DAAQ6.py ===
Given Array: [5, 2, 4, 7, 1, 3, 2, 6]
Sorted Array: [1, 2, 2, 3, 4, 5, 6, 7]
>>>
```