Algorithm Complexity

Mustafa Hajij
MSDS program
University of San Francisco

Outline

- 1. Introduction: what is an algorithm? What is algorithm complexity and why is it important to analyze it?
- 2. Time Complexity: how long does it take given the size of the input
- 3. Data Structures : Relationship between data structures and algorithm complexity, and how the choice of data structure impacts performance
- 4. Practical Considerations: Real-world considerations, including constant factors and hidden constants.
- 5. Conclusion

What is an algorithm?

```
def make_pancake ():
    print("Mix ingredients")
    print("Heat a pan")
    print("Pour batter")
    print("Flip the pancake")
    print("Cook until done")

# Example usage:
make_pancake()
```

This algorithm prints the main steps to make a pancake

What is an algorithm?

This algorithm sorts a list of numbers

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(my_list)
print("Sorted array:", my_list)
```

What is an algorithm?

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

# Example usage:
sorted_list = [11, 12, 22, 25, 34, 64, 90]
target_value = 25
result = binary_search(sorted_list, target_value)
print(f"Index of {target_value}:", result)
```

This algorithm returns 1 if target is in arr and -1 otherwise

UNIVERSITY OF SAN FRANCISCO

What is an algorithm?

This algorithm returns of the GCD of a and b.

```
def euclidean_gcd(a, b):
    while b:
        a, b = b, a % b
    return a

# Example usage:
num1, num2 = 48, 18
gcd_result = euclidean_gcd(num1, num2)
print(f"GCD of {num1} and {num2}:", gcd_result)
```

What is an algorithm?

This estimate the value of pi

```
import random

def monte_carlo_pi(num_samples):
    inside_circle = 0

    for _ in range(num_samples):
        # Generate random points within the unit square
        x, y = random.uniform(-1, 1), random.uniform(-1, 1)
        distance = x**2 + y**2

        # Check if the point is inside the unit circle
        if distance <= 1:
            inside_circle += 1

    # Estimate Pi based on the ratio of points inside the circle to the total points
    pi_estimate = (inside_circle / num_samples) * 4
    return pi_estimate

# Example usage:
num_samples = 100000
estimated_pi = monte_carlo_pi(num_samples)
print(f"Estimated value of Pi using Monte Carlo Simulation: {estimated_pi}")
```

What is an algorithm ?

- An algorithm is a systematic, step-by-step set of instructions or a clear plan that outlines how to perform a specific task.
- It provides a well-defined finite sequence of actions that, when followed, leads to a desired outcome.
- It operates on input data, transforming it through a series of well-defined steps into the desired output.

Algorithm Time Complexity

- Time complexity represents the amount of time an algorithm takes to complete its execution as a function of the input size.

- It provides an upper bound on the growth rate of the running time concerning the input size.

- Commonly expressed using Big O notation (e.g., ($O(1)$), ($O(\log n)$), ($O(n)$), ($O(n^2)$)).

Big O notation

Big O notation is a way to describe the upper bound or worst-case scenario of the growth rate of an algorithm's time or space complexity in relation to the size of the input.

Significance : Big O notation provides a concise way to express how the performance of an algorithm scales with the size of the input. It helps in comparing and analyzing algorithms in terms of their efficiency and scalability.

Big O notation

1. Linear Time Complexity (O(n)):

In a simple linear function:

$$f(n) = 3n + 5$$

In Big O notation, we ignore the coefficient term (5) and the constant factor (3):

O(n).

This is because the linear term dominates as n becomes large.

Big O notation

2. Quadratic Time Complexity (O(n^2)):

In a quadratic function:

$$f(n) = 2n^2 + 3n + 1$$

In Big O notation, we ignore the lower-order terms (3n + 1) and the constant factor (2):

O(n^2)

The quadratic term dominates as n becomes large.

Big O notation

3. Logarithmic Time Complexity (O(\log n)):

In a logarithmic function:

$$f(n) = 4 \log n + 2$$

In Big O notation, we ignore the constant factor (4) and the constant 2 :

O(log n)

The logarithmic term dominates as n becomes large.

When it comes to studying complexity (running time or memory), We care about asymptotic behavior

- Think about the following : imagine *n* getting very big and the worst-case input scenario, how long is it gonna take ? And how much space is it gonna take to perform the computations?
- Therefore, ignore constants, keep only most important terms:
  - $f(n) = 2n$ implies $O(n)$
  - $f(n) = n^3 + kn^2 + nlogn$ implies $O(n^3)$
  - $f(n) = k$ for constant k implies $O(1)$
- Example., $3n!$ and $10n!$ are the same asymptotically

Big O notation

1. $f(n) = 5 \Rightarrow O(1)$
2. $f(n) = 2n + 1 \Rightarrow O(n)$
3. $f(n) = 3n^2 + 2n + 1 \Rightarrow O(n^2)$
4. $f(n) = \log n \Rightarrow O(\log n)$
5. $f(n) = \mathrm{sqrt}(n) \Rightarrow O(\mathrm{sqrt}(n))$
6. $f(n) = n * \log n \Rightarrow O(n * \log n)$
7. $f(n) = 2^n \Rightarrow O(2^n)$
8. $f(n) = n! \Rightarrow O(n!)$
9. $f(n) = n^{1/3} \Rightarrow O(n^{1/3})$
10. $f(n) = \frac{1}{2}n\hat{}2 + 3n \Rightarrow O(n^2)$
11. $f(n) = e^n \Rightarrow O(e^n)$
12. $f(n) = n\hat{}2 + \log n \Rightarrow O(n^2)$
13. $f(n) = n * 2\hat{}n \Rightarrow O(n * 2^n)$
14. $f(n) = 2n^3 + 5n^2 + 3 \Rightarrow O(n^3)$
15. $f(n) = 4^n + n^3 \Rightarrow O(4^n)$
16. $f(n) = \frac{1}{2}n^2 + 3n + 1 \Rightarrow O(n^2)$
17. $f(n) = 100n + \log n \Rightarrow O(n)$
18. $f(n) = \frac{1}{2}n^3 + 3n^2 \Rightarrow O(n^3)$
19. $f(n) = n^{2.5} + n \Rightarrow O(n^{2.5})$
20. $f(n) = n^2 - n \Rightarrow O(n^2)$

UNIVERSITY OF SAN FRANCISCO

Algorithm Time Complexity

```
def constant_operation(num):
    # Constant Operation: O(1) complexity
    return num * 2 # O(1)

# Example usage:
input_value = 5
result = constant_operation(input_value)
print(f"Constant Operation Result: {result}")
```

Explanation of O(1): No matter how large the input (num) is, the running time remains constant. The growth rate is not dependent on the size of the input.

Algorithm Time Complexity

```
def linear_sum(arr):
    # O(n) - Linear time complexity
    result = 0
    for num in arr:
        result += num
    return result

# Example usage:
my_list = [1, 2, 3, 4, 5]
result_sum = linear_sum(my_list)
print(f"Result Sum: {result_sum}")
```

Explanation of O(n): The running time grows linearly with the size of the input array (arr). If the array has n elements, the algorithm takes approximately n steps.

Algorithm Time Complexity

```
def bubble_sort(arr):
    # Bubble Sort: O(n^2) complexity

    n = len(arr)  # O(1)
    for i in range(n):  # O(n) - Outer loop runs 'n' times
        for j in range(0, n - i - 1):  # O(n) - Inner loop runs 'n-i-1' times
            if arr[j] > arr[j + 1]:  # O(1)
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # O(1)

# Example usage:
my_list = [4, 2, 7, 1, 9, 5]
bubble_sort(my_list)
print("Bubble Sort Result:", my_list)
```

Compute complexity following our process

Require: Input X with $|X| = n$

```
sum = 0
for i = 1 to n do
    for j = 1 to n do
        sum ← sum + 1
    end for
end for
for k = 1 to n do
    X_k ← k
end for
return X
```

1. Identify key size indicator
2. Define *T*(*n*) = …
3. Reduce *T*(*n*) to closed form
4. *O*(*n*) is asymptotic behavior of *T*(*n*)

Compute complexity following our process

```
Require: Input X with |X| = n
sum = 0
for i = 1 to n do
    for j = 1 to n do
        sum ← sum + 1
    end for
end for
for k = 1 to n do
    X_k ← k
end for
return X
```

1. Identify key size indicator: n
2. $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 2 + \sum_{k=1}^{n} 1$
3. $T(n) = 2n^2 + n$ (closed form)
4. $O(n^2)$ asymptotic behavior

Compute complexities for these too

```
sum1 = 0
for i = 1 to n do
    for j = 1 to n do
        sum1 ← sum1 + 1
    end for
end for
sum2 = 0
for i = 1 to n do
    for j = 1 to i do
        sum2 ← sum2 + 8
    end for
end for
```

- Identify key size indicator: n
- $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 2 + \ldots$
- $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 2 + \sum_{i=1}^{n} \sum_{j=1}^{i} 2$
- $T(n) = 2n^2 + \sum_{i=1}^{n} 2i$
- $T(n) = 2n^2 + 2\sum_{i=1}^{n} i$
- $T(n) = 2n^2 + 2n(n+1)\ /\ 2$
- $T(n) = 2n^2 + 2n^2/2 + 2n/2 = 3n^2 + n$
- $O(n^2)$ asymptotic behavior

Example

function func(n)

$x \leftarrow 0$;

for $i \leftarrow 1$ **to** n **do**

    for $j \leftarrow 1$ **to** i **do**

        $x \leftarrow x + (i - j)$;

    end

end

return (x);

Example

```
function func(n)
x ← 0;
for i ← 1 to n do
    for j ← 1 to i do
        x ← x + (i − j);
    end
end
return (x);
```

Complexity : $O(n^2)$

Example

```
function func(n)
x ← 0;
i ← 7;
while (i ≤ n) do
    x ← x + i;
    i ← i + 3;
end
return (x);
```

Example

```
function func(n)
x ← 0;
i ← 7;
while (i ≤ n) do
    x ← x + i;
    i ← i + 3;
end
return (x);
```

Complexity : $O(n)$

Example

```
function func(n)
if (n > 100000) then return (0);
x ← 0;
for i ← 1 to n do
    for j ← 1 to n do
        x ← x + (i − j);
    end
end
return (x);
```

Example

function func(n)

if ($n > 100000$) **then return** (0);

$x \leftarrow 0$;

for $i \leftarrow 1$ **to** n **do**

  for $j \leftarrow 1$ **to** n **do**

    $x \leftarrow x + (i - j)$;

  end

end

return (x);

Complexity : $O(1)$

Example

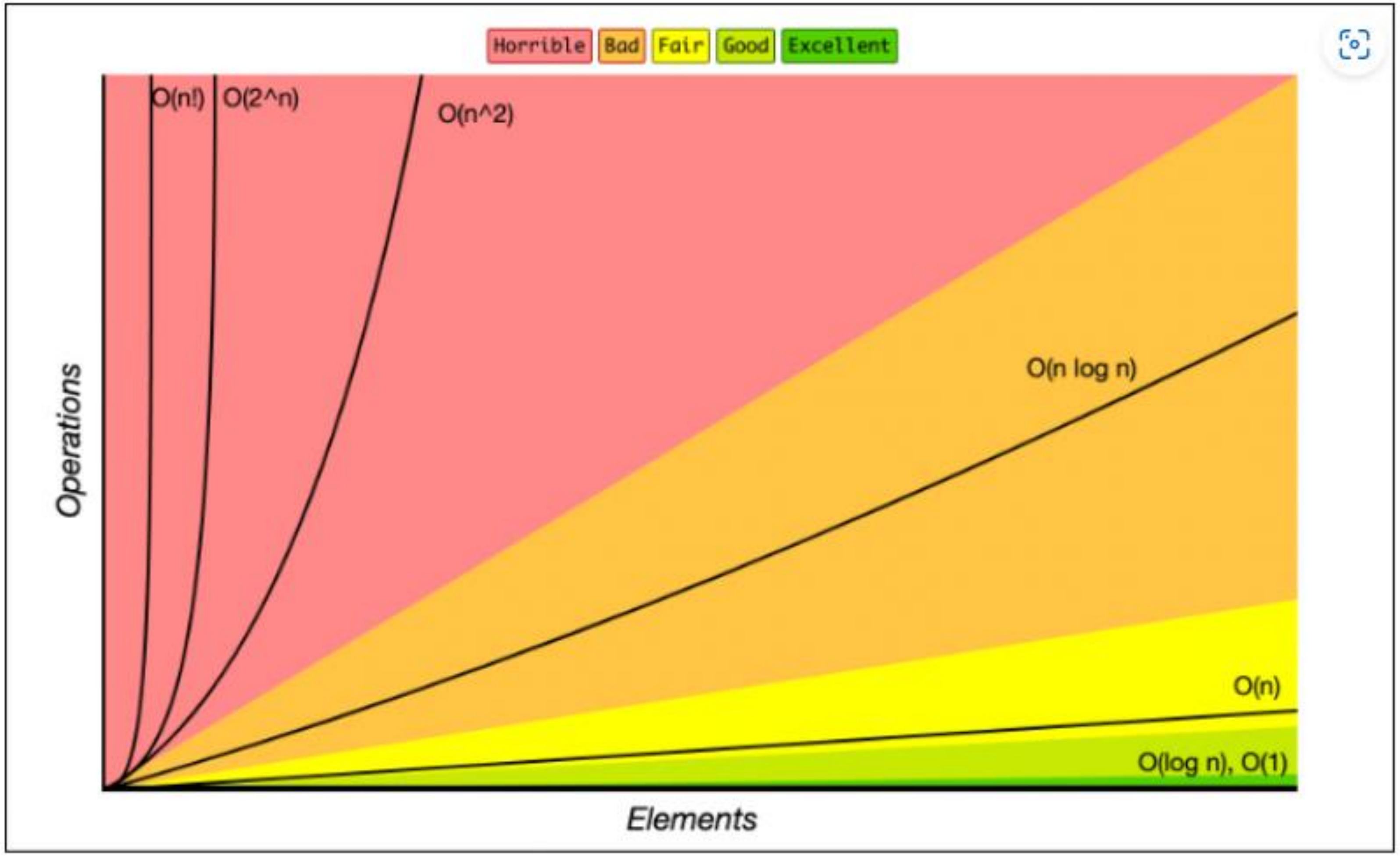
```
function func(n)
if (n < 100000) then return (0);
x ← 0;
for i ← 1 to n do
    for j ← 1 to n do
        x ← x + (i − j);
    end
end
return (x);
```

Example

```
function func(n)
if (n < 100000) then return (0);
x ← 0;
for i ← 1 to n do
    for j ← 1 to n do
        x ← x + (i − j);
    end
end
return (x);
```

Complexity : $O(n^2)$

How does it look visually

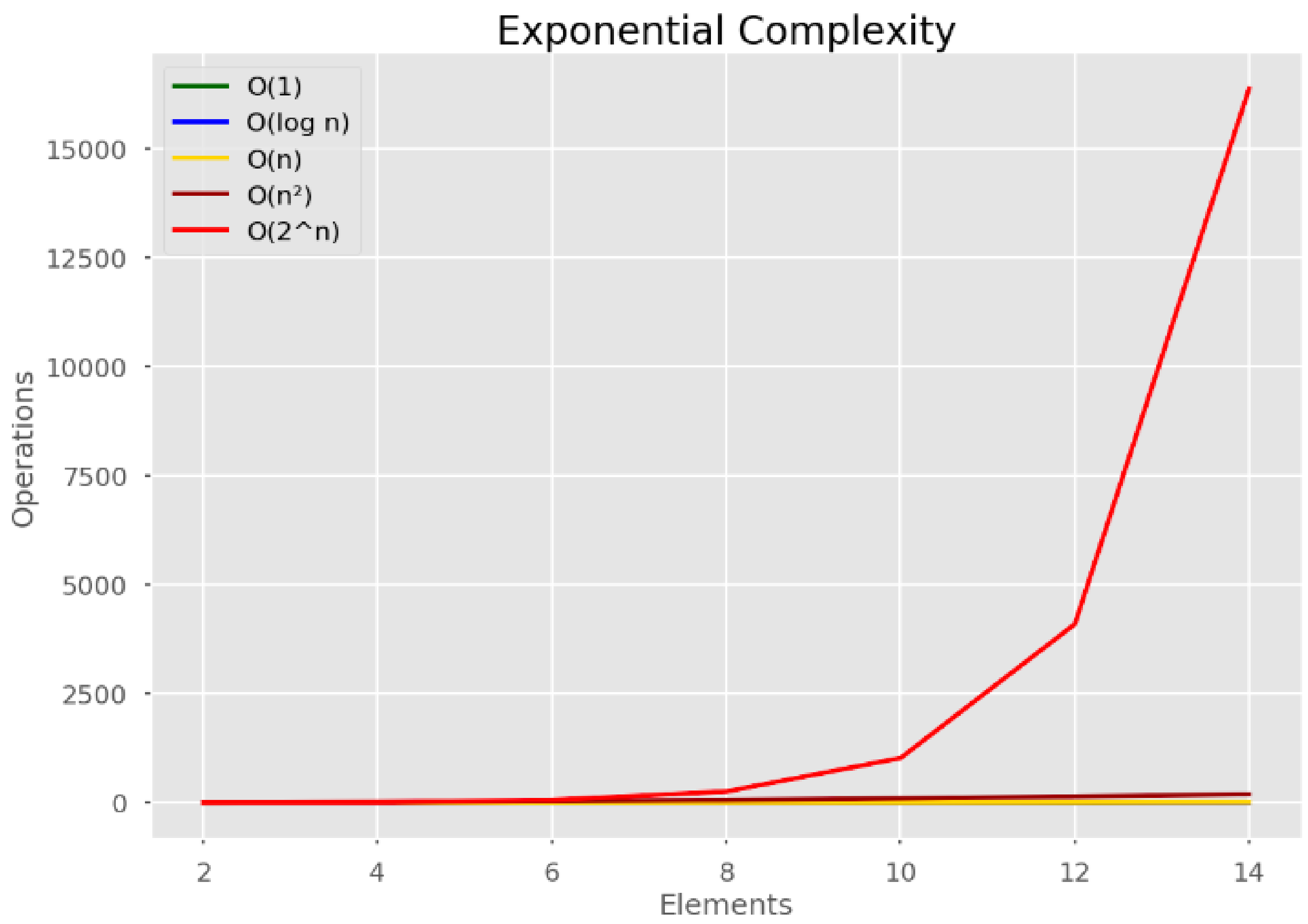


Ref : Big O Cheat Sheet – Time Complexity Chart (freecodecamp.org)

UNIVERSITY OF SAN FRANCISCO

How does it look visually

Exponential Time



[All You Need to Know About Big O Notation [Python Examples] (skerritt.blog)](https://skerritt.blog)

Recursive algorithms complexity

- Compute the complexity $T(0) = 0$, with $T(n) = 1 + T(n-1)$.

Recursive algorithms complexity

- Compute the complexity $T(0) = 0$, with $T(n) = 1 + T(n-1)$.

$T(n) = 1 + T(n-1)$
$T(n) = 1 + 1 + T(n-2)$
$T(n) = 1 + 1 + 1 + T(n-3) = n + T(n-n) = n + T(0) = n + 0 = n$

Recursive algorithms complexity

- Question :

Ask how many times you can divide n by 2?

Recursive algorithms complexity

- Question :

Ask how many times you can divide n by 2?

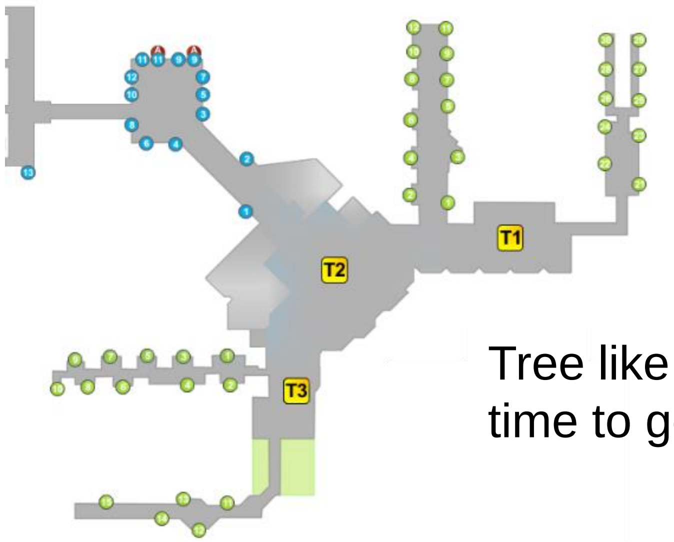
log(n) times.. Why ?

Recursive algorithms complexity

- Question :

Ask how many times you can divide n by 2?

log(n) times.. Why ?



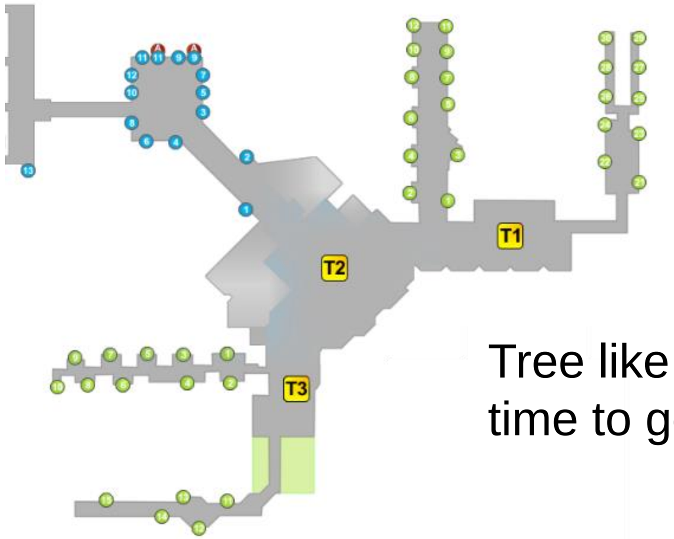
Tree like structures are common in airports design : save time to go to the destination gate.

Recursive algorithms complexity

- Question :

Ask how many times you can divide n by 2?

log(n) times.. Why ?



Tree like structures are common in airports design : save time to go to the destination gate.

We will see the relation between trees and divide by two later!

Recursive algorithms complexity

- $T(1) = 0$
  $T(n) = 1 + T(n/2)$
  $= 1 + 1 + T(n/4)$
  $= 1 + 1 + 1 + T(n/2^3)$

Recursive algorithms complexity

- $T(1) = 0$
  $T(n) = 1 + T(n/2)$
  $= 1 + 1 + T(n/4)$
  $= 1 + 1 + 1 + T(n/2^3)$

stop when 2^i reaches n, at T(n/n)=T(1)

Recursive algorithms complexity

- T(1) = 0
  T(n) = 1 + T(n/2)
  = 1 + 1 + T(n/4)
  = 1 + 1 + 1 + T(n/2^3)

stop when 2^i reaches *n*, at T(n/n)=T(1)

How many 1's are we gonna have?

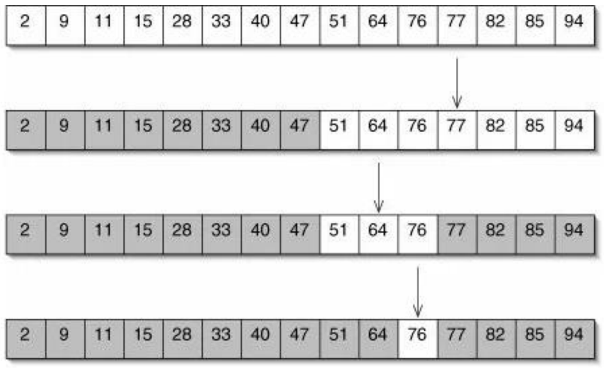
We stop after k steps, and exactly when $\frac{n}{2^k} = 1$.
This implies that $k = \log(n)$

Binary search

```
def binary_search(arr, target):
    """
    Binary Search Algorithm

    Parameters:
    - arr: A sorted array of elements.
    - target: The element to search for.

    Returns:
    - Index of the target element if found, else -1.
    """
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = (low + high) // 2  # Calculate the middle index

        if arr[mid] == target:
            return mid  # Target found, return the index
        elif arr[mid] < target:
            low = mid + 1  # Target is in the right half
        else:
            high = mid - 1  # Target is in the left half

    return -1  # Target not found
```

```
# Example usage:
sorted_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target_value = 7
result_index = binary_search(sorted_array, target_value)
print(f"Index of {target_value} in the array: {result_index}")
```

- In each iteration of the `while` loop, the search space is effectively halved by adjusting `low` or `high` based on the comparison with the middle element.
- The time complexity of binary search is (O(log n), where n is the number of elements in the sorted array.
- This logarithmic complexity stems from the fact that each iteration reduces the search space by half.



$$1 = n/2^x$$
$$2^x = n$$

take log base 2 both side:

$$\log_2(2^x) = \log 2\, n$$
$$x * \log_2(2) = \log_2 n$$
$$\boldsymbol{x * 1 = \log_2 n}$$

Recursive algorithms complexity

```
def T(n): # for n>=1
    if n <= 1: return 0
    counting = 0
    while n > 0:
        n = n // 2
        counting += 1
    return counting -1
```

What is the complexity here?

Recursive algorithms complexity

```
def T(n): # for n>=1
    if n <= 1: return 0
    counting = 0
    while n > 0:
        n = n // 2
        counting += 1
    return counting -1
```

What is the complexity here?

Answer : O(log(n)). Why?

Recursive algorithms complexity

Sometimes it is useful to “draw” a tree associated with the recursion and use that tree to compute the complexity

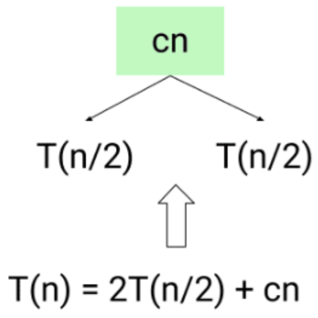
$$T(n) = 2T(n/2) + cn$$

Analysis of Recursion in Data Structures and Algorithms (enjoyalgorithms.com)

UNIVERSITY OF SAN FRANCISCO

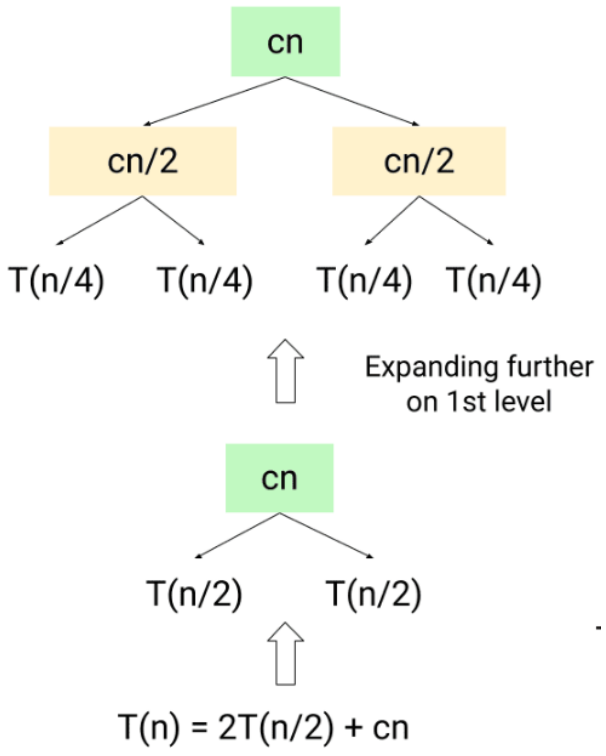
Recursive algorithms complexity

Sometimes it is useful to "draw" a tree associated with the recursion and use that tree to compute the complexity



Recursive algorithms complexity

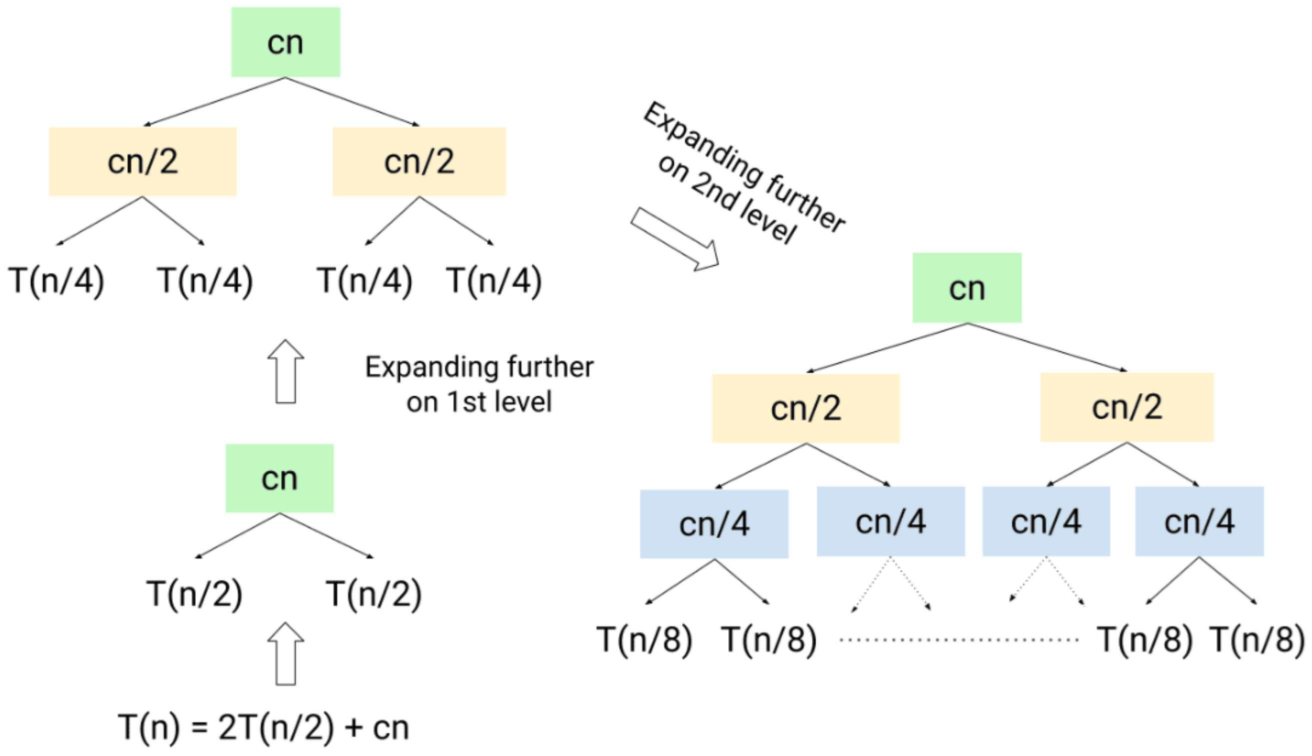
Sometimes it is useful to “draw” a tree associated with the recursion and use that tree to compute the complexity



[Analysis of Recursion in Data Structures and Algorithms (enjoyalgorithms.com)](https://www.enjoyalgorithms.com)

Recursive algorithms complexity

Sometimes it is useful to “draw” a tree associated with the recursion and use that tree to compute the complexity



[Analysis of Recursion in Data Structures and Algorithms (enjoyalgorithms.com)](https://www.enjoyalgorithms.com)

Big O for some common recurrence relations

$T(n) = T(n-1) + c$	$O(n)$
$T(n) = T(n/2) + c$	$O(\log n)$
$T(n) = 2 * T(n/2) + c_a n + c_b$	$O(n \log n)$
$T(n) = 2 * T(n-1) + c$	$O(2^n)$

Data Structure Operation Complexity

Arrays:

Definition: An array is a collection of elements, each identified by an index or a key.

Time Complexity:

-Access (Read/Write): O(1) - Accessing an element in an array by index is constant time.

- Insertion/Deletion at End: O(1) - Adding or removing an element at the end of an array is constant time.
- Insertion/Deletion at Arbitrary Position: O(n) - Inserting or deleting an element at an arbitrary position requires shifting elements and takes linear time.

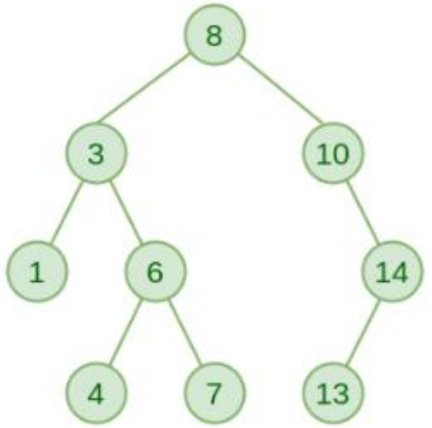
Data Structure Operation Complexity

Trees (Binary Search Trees):

Definition: A binary search tree is a hierarchical data structure where each node has at most two children, and elements are arranged such that the left subtree contains elements less than the node, and the right subtree contains elements greater than the node.

Time Complexity:

•**Search/Insert/Delete:** $O(\log n)$ (if the tree is balanced) - In a balanced binary search tree, these operations are logarithmic in the number of elements.



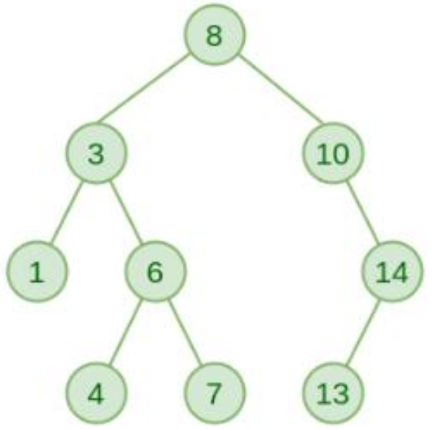

```
p = root
while p is not None:
    if p.value==x: return p
    if x < p.value: p = p.left
    else: p = p.right
```

What is the max height of this tree?
What is the min height of this tree?

Data Structure Operation Complexity

Trees (Binary Search Trees):

Definition: A binary search tree is a hierarchical data structure where each node has at most two children, and elements are arranged such that the left subtree contains elements less than the node, and the right subtree contains elements greater than the node.



Time Complexity:

•**Search/Insert/Delete:** $O(\log n)$ (if the tree is balanced) - In a balanced binary search tree, these operations are logarithmic in the number of elements.

```
p = root
while p is not None:
    if p.value==x: return p
    if x < p.value: p = p.left
    else: p = p.right
```

Search complexity : $T(n) = 1 + T(n/2)$

What is the max height of this tree?
What is the min height of this tree?

Data Structure Operation Complexity

Hash Tables (dictionaries in python):

Definition: A hash table is a data structure that maps keys to values, and it uses a hash function to compute an index into an array.

Time Complexity:

•**Insert/Search/Delete:** *O*(1) (on average) - If the hash function distributes elements uniformly, these operations are constant time on average.

Practical consideration

1. Constant Factors:

- Asymptotic analysis focuses on the growth rate of algorithms, but it often ignores constant factors. In practice, these constants can significantly impact performance.
- An algorithm with a lower theoretical complexity may have a higher constant factor, making it slower for small input sizes.

2. Hidden Constants:

- Some algorithms may have hidden constants that are not apparent in the asymptotic notation. These constants can be influenced by implementation details, hardware architecture, and language choice.
- For example, algorithms with larger constant factors might be faster on certain hardware architectures.

3. Caching and Memory Access:

- Algorithms that make better use of caching and minimize memory access can have better real-world performance. Cache misses and inefficient memory access patterns can lead to performance bottlenecks.
- Consideration of data locality and cache efficiency can impact the practical performance of algorithms.

4. Input Characteristics:

- The nature of the input data can affect the algorithm's performance. Some algorithms may perform well on certain types of input and poorly on others.
- Real-world data distribution and patterns should be considered. Algorithms should be chosen or adapted based on the expected input characteristics.

Practical consideration

5. Parallelism and Concurrency:

- Modern hardware often includes multiple cores, and parallel algorithms can exploit parallelism for improved performance.
- Algorithms designed to take advantage of parallel processing or concurrency may outperform their serial counterparts.

6. I/O Operations:

- Algorithms that involve I/O operations (reading/writing to disk, network operations) can have different performance characteristics compared to purely computational algorithms.
- Optimizing I/O-bound algorithms may require different considerations than CPU-bound algorithms.

7. Practical Constraints:

- Some algorithms may have theoretical advantages but may not be practical due to real-world constraints. For example, an algorithm with lower time complexity but higher space complexity might not be suitable for memory-constrained environments.

8. Programming Language and Compiler:

- The choice of programming language and compiler can impact the performance of an algorithm. Different languages and compilers may optimize code differently.
- Language features, libraries, and runtime environments can also affect the practical efficiency of an algorithm.

9. Implementation Quality:

- The quality of the algorithm's implementation, code optimizations, and choice of data structures can significantly impact real-world performance.
- A well-optimized implementation can sometimes outperform a theoretically more efficient algorithm with a suboptimal implementation.

Conclusion

In summary, algorithmic complexity analysis provides a concise framework for evaluating efficiency, considering both theoretical growth rates and practical considerations.

1. Growth Rate Analysis:
   - Focuses on understanding how an algorithm's resource consumption (time or space) grows with input size.
   - Utilizes Big O notation for expressing the upper bound of this growth rate.

2. Time vs. Space Complexity:
   - Time complexity measures execution time based on input size.
   - Space complexity gauges memory usage relative to input size.
   - Balancing these complexities is crucial for efficient algorithm design.

3. Asymptotic Notations:

4. Practical Considerations:
   - Considers real-world factors like constant factors, hidden constants, and hardware constraints.
   - Practicality is essential for making informed algorithmic choices that align with specific application needs.