Data Structures and Algorithm Analysis

Data Structures and Algorithm Analysis serves as essential coursework in artificial intelligence, computer science, and data science, aiming to thoroughly explain the basic data structures and algorithms along with methods to evaluate their effectiveness. The module, delivered through lectures, hands-on labs, and problem solving assignments, ensures an understanding of how to design, execute, and evaluate data structures and algorithms to efficiently solve diverse computational challenges.

The educational goal in the study of *data structures* is to understand a variety of data structures, including arrays, linked lists, stacks, queues, trees, and graphs. Data structures are methods for organizing and storing data on a computer to enable efficient use. Arrays consist of a set of elements, each identified by an index or key. Linked lists are composed of a series of elements, each linking to the subsequent one. Stacks operate on a Last In, First Out (LIFO) principle, where elements are both added and removed from the same end. Queues function on a First In, First Out (FIFO) basis, with elements being added at the back and removed from the front. Trees are hierarchically structured with nodes that are interconnected by edges. Graphs consist of a set of nodes (vertices) connected by edges.

The goal of learning *Algorithm Analysis* is to understand methods for evaluating the performance of algorithms, focusing on both time complexity and space complexity. This analysis assesses how efficiently algorithms perform by examining their time and space complexities. Time complexity refers to the time required for an algorithm to execute, depending on the size of the input, with typical notation such as $O(n), O(\log n), O(n^2)$. Space complexity evaluates the memory usage required by an algorithm, also based on input size, with notations such as $O(1), O(n), O(n^2)$.

Throughout this semester, the foundational techniques discussed in this module will be expanded on for *Algorithm Design Techniques*. This includes a variety of algorithm design methods such as divide-and-conquer, greedy algorithms, dynamic programming, and backtracking. Each of these techniques contributes to the development of efficient algorithms. Divide and conquer involves breaking down a problem into smaller subproblems, solving each recursively, and then merging the solutions. Greedy algorithms continuously make locally optimal choices with the aim of achieving a globally optimal solution. Dynamic Programming (DP) divides a problem into smaller, overlapping subproblems, solves each one once, and saves these solutions to prevent repetitive work. In recursion or reverse search, a systematic search is performed through all possible solutions to determine the best solution, often employing recurrence.

This document is an extension of the research and lecture notes completed at Johns Hopkins University, Whiting School of Engineering, Engineering for Professionals, Artificial Intelligence Master's Program, Computer Science Master's Program, and Data Science Master's Program.

Contents

1 Data Variables

Given a dataset of M variables and N dataset points (number of samples), a feature is one of the **independent** variables in a dataset and is also called a **predictor**. Generally, the input to a machine learning program is a column of a tabular dataset where each row (of N rows) is a dataset point in M dimensional space.

In our Jupyter notebooks, we will use the matrix $X_{N\times M}$ as the dataset symbol without the dependent variable (also called the label, category, class, or predicted variable), and we will store the dependent variable in the vector $y_{N\times 1}$.

X is a matrix. Its rows are data points, and its columns are the features. All classifiers in `scikit-learn` can understand this data format based on `numpy` 2-dimensional arrays.

Thus, for each input data point, we have $x \in \mathbb{R}^M$, and we have N data points to be used in our ML pipelines. We also have $y \in \mathbb{Z}^+$ in general as the category. As an example, if we have K categories, then $y \in \{k : 0 \leq k < K, k \in \mathbb{Z}^+\}$.

In the following X matrix, we have 3 features and 5 data points, i.e. $M = 3$ and $N = 5$. We also have 5 values for the dependent variable y.

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} \qquad \text{Such that in practice, } X = \begin{bmatrix} 4.9 & 3. & 1.4 \\ 4.7 & 3.2 & 1.3 \\ 4.6 & 3.1 & 1.5 \\ 5. & 3.6 & 1.4 \\ 5.4 & 3.9 & 1.7 \end{bmatrix}, y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 2 \end{bmatrix}$$

where the dependent variable y has levels from the alphabet $\Sigma = \{0, 1, 2\}$, i.e. there are 3 categories in the given dataset.

The predicted variable or label is the **dependent** variable, dependent on the **independent variables** or features. This dependence can sometimes be high and sometimes very low, depending on the dataset or the nature of the problem (again, the dataset expresses this). If there is no correlation (or fully independent), then the dataset may not be suitable for the problem at hand, and/or we may have to remove that feature from the dataset.

As an example, for numerical variables, the Pearson correlation coefficient of two variables x (lower case x, a feature) and y is defined as:

$$r_{xy} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}}$$

where $\bar{x}$ and $\bar{y}$ are sample means.

Ideally, we need a good correlation (close to 1 or -1) between the independent and dependent (predicted) variables so our ML model would actually work.

2 Data types

- Numerical - Can be integer $\mathbb{Z}$ or floating point $\mathbb{R}$. Generally, it is safe to convert all numerical variables to floating point variables
- Nominal - The variable values are drawn from a finite set of **levels** or from an alphabet Σ
- Ordinal type are nominals with natural order, such as small, medium, big
- Binary - The variable values can be either `0` or `1` (or, `False` or `True`). Some algorithms work fast on this kind of values, especially constrained optimization-related methods
- String - May not be used directly unless the ML program (preprocessing) knows how to deal with it
- Date - May not be used directly unless the ML program (preprocessing) knows how to deal with it. It might be a good idea to convert (or map) dates to some integer numbers - for example, Excel handles dates in this manner
- More complicated features, e.g., a DNA sequence (sequence of A,C,G,T letters) - Other, simpler, features need to be extracted from the input sequence so that this higher-level feature can be used in an ML program

2.1 Nominal to Numerical Conversion

One possible way of converting nominal variables to numerical is **one-hot encoding**:

1. During preprocessing, count the number of levels in the set of possible levels a nominal variable v_{nom} takes. Such as, L different levels, $k = 1, \ldots, L$.
2. Create L binary variables for that nominal variable v_{nom} where each row will have a binary zero for $L - 1$ binary variables except for the j^{th} level which corresponds to the level-j when v_{nom} takes a value of level-j.

Following the above procedure, a nominal variable with a cardinality of L results in L many binary variable creation (and dropping the original nominal variable itself). In other words, each unique level of that nominal variable is mapped to a binary variable. Note that, for the sake of this representation, storage space is wasted.

Also, observe that the one-hot encoded variables are like the Cartesian basis of linear algebra, e.g., a feature one-hot encoded to 3 dimensions of x, y, z.

Conversion of nominal variables to numerical is an important step for many numerical-only classifiers, such as neural networks, support vector machines, and linear regression.

Note that using `sklearn.preprocessing.LabelEncoder` will impose ordinal numerical values, which may not be correct and cause bias. Therefore, it should be avoided. Instead, use `OneHotEncoder`.

2.2 Example One-hot Encoding

The nominal variable T take **levels** from the $\Sigma = \{\text{low}, \text{medium}, \text{high}\}$. Numerical conversion involves each unique level being mapped to one of the T_i binary vectors.

2.3 Numerical to Nominal Conversion

Generally, histograms, binning, and bin boundaries are used to group numerical values into levels or one-hot encoded variables.

3 Online Dataset Sources

The **UCI KDD** online repository has various datasets that can be used for analysis, machine learning, and several application fields, such as GIS, cybersecurity, NLP, etc. The origin of some datasets goes back more than 20 years, sourced from competitions, challenges, grants, etc. Researchers and students use these datasets and share their experiences using a common platform. Source: UCI Knowledge Discovery in Databases Archive https://kdd.ics.uci.edu/

The **Kaggle** data repository has various datasets used for Kaggle competitions. The website also has tools to examine the features on-site. This source is one of the largest. Source: https://www.kaggle.com/da

The **KDnuggets** is another web page that encompasses almost everything (posts, news, datasets, tutorials, forums, webinars, software, etc.) relevant to machine learning and data analysis. Source: KDnuggets Datasets for Data Mining and https://www.kdnuggets.com/datasets/index.html

The U.S. government's open data website **Data.gov** is another resource that provides access to datasets published by agencies across the federal government. Source: https://data.gov/

The rest of the notes will demonstrate three different datasets from these repositories.

1. UCI KDD archive → 1990 US Census data
2. Kaggle → Graduate Admissions data
3. Kaggle → The Human Freedom Index data

Download the data files from UCI KDD website and Kaggle website (by registering to Kaggle -using a disposable email address if necessary).

Important Note: About physical dataset file shared among teams. Comparing machine learning models and measuring performances for model selection depends heavily on the input dataset. Thus, if a comparison between models and comparison among different experiments or teams' results are at hand, then the dataset shared among teams or different sets of experiments **must** be the same dataset. Moreover, to ensure validity, the same file should be shared among multiple teams or between different model pipelines.

4 Data Structures

This section presents various topics related to data structures for effective data work and analysis. Lists and Arrays are important for storing and manipulating ordered collections of data. Concepts covered include indexing, slicing, and operations like append, insert, and delete. Understanding multidimensional arrays is crucial for working with matrices and numerical data. Dictionaries (Hash Maps) are crucial for storing key-value pairs, enabling quick lookups and data organization. Stacks and Queues are useful for managing data in a last-in-first-out (LIFO) or first-in-first-out (FIFO) manner, respectively. Knowledge of linked lists, whether singly or doubly linked, can be beneficial for various data manipulation tasks. Trees and Graphs are essential for understanding tree and graph data structures, which are used in tasks like decision tree-based algorithms and network analysis. Heaps are commonly used for priority queues and can be advantageous in optimization problems. Sets are helpful for storing unique elements and performing set operations such as union, intersection, and difference. Tuples are similar to lists but are immutable, making them suitable for storing fixed data structures. Data frames, commonly used in libraries like Pandas in Python, are tabular data structures that are central to data manipulation and analysis. Understanding how hashing functions work and their applications in data retrieval and data integrity is important for Hashing. Tries are tree-like structures used in applications such as autocomplete and spell checking. These data structure topics form the foundation for various data processing, analysis, and machine learning tasks. To gain a deeper understanding, it is recommended to research these topics in more detail using Python-based data structure frameworks and datasets from online resources such as Kaggle, Data.gov, and open-source repositories.

4.1 Lists and Arrays

Lists and arrays are fundamental data structures in Python used extensively in data science projects.

Lists are ordered collections of items. They can contain elements of different data types and are mutable, which means you can modify their contents.

Creating a List

```
# Creating a list
my_list = [1, 2, 3, 4, 5]

# Accessing elements
print(my_list[0])  # Access the first element (index 0)

# Modifying elements
my_list[2] = 10  # Modify the third element (index 2)

# Adding elements
my_list.append(6)  # Add an element to the end of the list

# Removing elements
my_list.remove(4)  # Remove an element by value
```

> Output:
>
> ```
> 1
> ```

NumPy as described in the Python refresher is a good library for numerical operations, and it provides arrays that are more efficient for numerical computations compared to lists.

Creating a NumPy Array

```
# Importing NumPy
import numpy as np

# Creating a NumPy array
my_array = np.array([1, 2, 3, 4, 5])

# Accessing elements
print(my_array[0])  # Access the first element (index 0)

# Modifying elements
my_array[2] = 10  # Modify the third element (index 2)

# Basic array operations
sum_result = np.sum(my_array)  # Sum of all elements
mean_result = np.mean(my_array)  # Mean of all elements

# Generating arrays
zeros_array = np.zeros(5)  # Array of zeros
ones_array = np.ones(5)    # Array of ones

# Array operations
result = my_array * 2  # Multiplying all elements by 2
```

> Output:
>
> ```
> 1
> ```

Creating a Multidimensional Array

```
# Importing NumPy
import numpy as np

# Creating a 2D NumPy array
matrix_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing elements
print(matrix_2d[0, 1])  # Access element at row 0, column 1

# Slicing
```

```
row_1 = matrix_2d[1, :]  # Get the entire second row
column_2 = matrix_2d[:, 2]  # Get the entire third column

# Matrix operations
matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])

# Element-wise addition
result_addition = matrix_a + matrix_b

# Matrix multiplication
result_multiplication = np.dot(matrix_a, matrix_b)

# Transpose
matrix_transpose = np.transpose(matrix_a)

# Shape and dimensions
shape = matrix_2d.shape  # Get the shape of the 2D array
dimensions = matrix_2d.ndim  # Get the number of dimensions

# Generating multidimensional arrays
identity_matrix = np.eye(3)  # 3x3 identity matrix
random_matrix = np.random.rand(2, 2)  # 2x2 random matrix

# Reshaping arrays
flat_array = np.array([1, 2, 3, 4, 5, 6])
reshaped_array = flat_array.reshape(2, 3)  # Reshape to a 2x3 array
```

> Output:
>
> ```
> 2
> ```

4.2 Dictionaries

Dictionaries consist of pairs of keys and values, with each key being distinct and linked to a specific value. They are highly adaptable and extensively utilized in the field of data science for a range of applications, including the storage of metadata or the conversion of categorical data into numerical representations.

Creating a Dictionary

```
# Creating a dictionary
student_info = {
    "name": "John Doe",
    "age": 26,
    "major": "Data Science",
    "GPA": 3.85
```

```
}

# Accessing values using keys
name = student_info["name"]
age = student_info["age"]
major = student_info["major"]
GPA = student_info["GPA"]

# Modifying values
student_info["GPA"] = 3.9  # Update GPA

# Adding new key-value pairs
student_info["university"] = "Johns Hopkins University"

# Checking if a key exists
if "GPA" in student_info:
    print("GPA is available in the dictionary.")

# Removing key-value pairs
del student_info["major"]

# Iterating through keys and values
for key, value in student_info.items():
    print(f"{key}: {value}")

# Dictionary comprehension
grades = {"Algorithms": 90.5, "Data Science": 85.9, "Data Engineering": 79.9}
passed_subjects = {subject: score for subject, score in grades.items() if score >= 80}

# Nested dictionaries
students = {
    "student1": {"name": "Alice", "age": 25},
    "student2": {"name": "Bob", "age": 28}
}

# Handling missing keys
# Get value or default if key is missing
subject_score = grades.get("Data Patterns and Representations", "N/A")

# Length of a dictionary
num_keys = len(student_info)

# Keys and values as lists
keys = list(student_info.keys())
values = list(student_info.values())
```

Output:

```
GPA is available in the dictionary.
name: John Doe
age: 26
GPA: 3.9
university: Johns Hopkins University
```

Dictionaries play a crucial role in data science as they are highly beneficial for various tasks such as data preprocessing, feature engineering, and data transformation. These tasks often require mapping and manipulating data based on specific criteria or metadata.

4.3 Stacks and Queues

Stacks and queues play a crucial role as data structures in Python and find applications in a wide range of data science assignments and projects.

Creating a Stack A stack is a type of linear data structure that adheres to the Last-In-First-Out (LIFO) principle. In a stack, elements are inserted and deleted from the same end, referred to as the "top" of the stack.

```python
# Create an empty stack
stack = []

# Push elements onto the stack
stack.append(1)
stack.append(2)
stack.append(3)

# Pop elements from the stack and assign the top element
popped_element = stack.pop()

# Check if the stack is empty
is_empty = len(stack) == 0

# Example of reversing a string using a stack
def reverse_string(input):
    stack = []
    for char in input:
        stack.append(char)
    reversed_string = ""
    while stack:
        reversed_string += stack.pop()
    return reversed_string

reversed_string = reverse_string("Hello World!")
```

```
# Display the reversed string
print("Reversed String:", reversed_result)
```

> Output:
>
> ```
> Reversed String: !dlroW ,olleH
> ```

Creating a Queue A queue is a different type of linear data structure that adheres to the principle of First-In-First-Out (FIFO). In a queue, elements are inserted at the rear end and removed from the front end.

```
# Import the deque class for implementing a queue
from collections import deque

# Create an empty queue
queue = deque()

# Enqueue three elements
queue.append(1)
queue.append(2)
queue.append(3)

# Dequeue the front element as assign it to a variable
dequeued_element = queue.popleft()

# Check for an empty queue
is_empty = len(queue) == 0

# Example of a task queue
def task_queue(queue1):
    while queue1:
        task = queue1.popleft()
        print("Elements:", task)

# Enqueue task queue
queue1 = deque(["Hello", "World", "!"])
task_queue(queue1)
```

> Output:
>
> ```
> Elements: Hello
> Elements: World
> Elements: !
> ```

Data science utilizes stacks and queues for a range of purposes, including executing graph traversal algorithms, organizing data in a particular sequence, and resolving problems that necessitate a Last-In-First-Out (LIFO) or First-In-First-Out (FIFO) methodology.

4.4 Linked Lists

A linked list is a type of data structure that is composed of nodes. Each node contains two components: data and a reference (or pointer) to the next node in the sequence. There are various types of linked lists, including singly linked lists where each node points to the next node, and doubly linked lists where each node points to both the next and previous nodes.

Creating a Linked List

```
# Define a Node class to represent individual elements in the linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Define a LinkedList class to manage the linked list
class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    def search(self, target):
        current = self.head
        while current:
            if current.data == target:
                return True
            current = current.next
        return False

    def delete(self, target):
        if not self.head:
```

```
            return

        if self.head.data == target:
            self.head = self.head.next
            return

        current = self.head
        while current.next:
            if current.next.data == target:
                current.next = current.next.next
                return
            current = current.next

# Creating a singly linked list
my_list = LinkedList()
my_list.append(1)
my_list.append(2)
my_list.append(3)

# Displaying the linked list
my_list.display()

# Searching for an element
result = my_list.search(2)

# Deleting an element
my_list.delete(2)
```

> Output:
>
> ```
> 1 -> 2 -> 3 -> None
> ```

Linked lists are valuable in the field of data science for tasks that require the storage of dynamic data or the manipulation of sequential data.

4.5 Trees and Graphs

Data science relies heavily on trees and graphs as essential data structures for a variety of tasks, such as decision tree algorithms, network analysis, and data modeling.

Creating a Tree A tree refers to a data structure that is organized in a hierarchical manner, where nodes are linked together by edges. Nodes can have multiple child nodes or none at all, but they always have a single parent node, with the exception of the root node. Trees have various applications, including decision trees and hierarchical data storage.

```
# Define a TreeNode class to represent individual nodes in the tree
class TreeNode:
    def __init__(self, data):
```

```
        self.data = data
        self.children = []

    def add_child(self, child_node):
        self.children.append(child_node)

# Create a tree
root = TreeNode("Root")
child1 = TreeNode("Child 1")
child2 = TreeNode("Child 2")
root.add_child(child1)
root.add_child(child2)

# Display the tree using a recursive function
def display_tree(node, depth=0):
    print("  " * depth + node.data)
    for child in node.children:
        display_tree(child, depth + 1)

display_tree(root)
```

> Output:
>
> ```
> Root
>   Child 1
>   Child 2
> ```

Creating a Graph A graph is a structure for storing data that includes nodes (also known as vertices) and edges that link these nodes together. Graphs are utilized for a range of data analysis purposes, such as network analysis and modeling social networks.

```
# Define a Graph class to represent a graph
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.graph:
            self.graph[vertex1].append(vertex2)
        if vertex2 in self.graph:
            self.graph[vertex2].append(vertex1)

    def display_graph(self):
```

```
        for vertex, neighbors in self.graph.items():
            print(f"{vertex}: {', '.join(neighbors)}")

# Create a graph
my_graph = Graph()
my_graph.add_vertex("A")
my_graph.add_vertex("B")
my_graph.add_vertex("C")
my_graph.add_edge("A", "B")
my_graph.add_edge("B", "C")

# Display the graph
my_graph.display_graph()
```

> Output:
>
> ```
> A: B
> B: A, C
> C: B
> ```

The utilization of trees and graphs is crucial in data science for the depiction and examination of diverse data structures and connections.

4.6 Heaps

A heap is a data structure based on a binary tree that has a specific property called the heap property. In a min-heap, for instance, the value of the parent node is either smaller or equal to the values of its children, guaranteeing that the smallest element is always located at the root.

Creating a Min-Heap

```
# Importing the heapq module for heap operations
import heapq

# Create an empty list to represent a min-heap
min_heap = []

# Insert elements into the min-heap
heapq.heappush(min_heap, 3)
heapq.heappush(min_heap, 1)
heapq.heappush(min_heap, 4)
heapq.heappush(min_heap, 2)

# Get the minimum element without removing it
min_element = min_heap[0]

# Pop the minimum element from the min-heap
min_value = heapq.heappop(min_heap)
```

```
# Display the min-heap
print(min_heap)  # The remaining elements will be reordered to maintain the heap property
```

> Output:
>
> ```
> [2, 3, 4]
> ```

Creating a Max-Heap

```
# Create an empty list to represent a max-heap
max_heap = []

# Insert elements into the max-heap (using negation to simulate a max-heap)
heapq.heappush(max_heap, -3)
heapq.heappush(max_heap, -1)
heapq.heappush(max_heap, -4)
heapq.heappush(max_heap, -2)

# Get the maximum element without removing it (negate the result)
max_element = -max_heap[0]

# Pop the maximum element from the max-heap (negate the result)
max_value = -heapq.heappop(max_heap)

# Display the max-heap
print(max_heap)  # The remaining elements will be reordered to maintain the heap property
```

> Output:
>
> ```
> [-3, -2, -1]
> ```

In the field of data science, heaps are frequently employed in algorithms such as Dijkstra's shortest path algorithm. They are also utilized in a range of optimization problems where the objective is to efficiently locate the minimum or maximum values.

4.7 Sets

A set is a collection of elements that are not arranged in any particular order and each element is unique. Sets are beneficial for carrying out set operations such as union, intersection, and difference. They are frequently employed in data science to manage distinct values and execute data manipulation tasks.

Creating a Set

```
# Creating a set
my_set = {1, 2, 3, 4, 5}

# Adding elements to a set
```

```
my_set.add(6)
my_set.add(7)

# Removing elements from a set
my_set.remove(4)

# Checking if an element is in the set
element_exists = 3 in my_set

# Iterating through a set
for item in my_set:
    print(item)

# Set operations
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

# Union of sets
union_result = set1.union(set2)

# Intersection of sets
intersection_result = set1.intersection(set2)

# Difference of sets
difference_result = set1.difference(set2)

# Subset check
is_subset = set1.issubset(set2)

# Superset check
is_superset = set1.issuperset(set2)
```

> Output:
>
> ```
> 1
> 2
> 3
> 5
> 6
> 7
> ```

Sets play a crucial role in data science, especially when it comes to tasks such as eliminating duplicates from datasets, carrying out set operations on categorical data, and identifying unique elements within data collections.

4.8 Tuples

A tuple is a sequence of elements that are arranged in a specific order and cannot be modified after creation. In the field of data science, tuples are employed when there is a requirement to store and retrieve a predetermined set of values.

Creating a Tuple

```
# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)

# Accessing elements by index
element = my_tuple[2]  # Access the third element (index 2)

# Iterating through a tuple
for item in my_tuple:
    print(item)

# Tuple packing and unpacking
name = "John"
age = 30
location = "New York"

# Packing values into a tuple
person_info = (name, age, location)

# Unpacking values from a tuple
name, age, location = person_info

# Length of a tuple
tuple_length = len(my_tuple)

# Concatenating tuples
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
combined_tuple = tuple1 + tuple2

# Slicing a tuple
sliced_tuple = my_tuple[1:4]  # Get elements from index 1 to 3

# Checking if an element exists in a tuple
element_exists = 3 in my_tuple

# Counting occurrences of an element in a tuple
count = my_tuple.count(3)

# Finding the index of an element in a tuple
```

```
index = my_tuple.index(4)

# Tuple with a single element (note the comma)
single_element_tuple = (1,)

# Immutability of tuples
# my_tuple[0] = 10  # This will result in an error since tuples are immutable
```

> Output:
>
> ```
> 1
> 2
> 3
> 4
> 5
> ```

Tuples are frequently employed in the field of data science to guarantee the preservation of a set of values throughout the entire data processing pipeline. They are also utilized to return multiple values from functions and to organize data in a predetermined format.

4.9 Data Frames

A data frame is a structured data representation with labeled axes (rows and columns), presented in a tabular format. It provides an efficient way to store and manipulate structured data. The Pandas library is commonly utilized in the field of data science for working with data frames.

Creating a Data Frame

```
# Importing the Pandas library
import pandas as pd

# Creating a data frame from a dictionary
data = {
    'Name': ['John', 'Lamar', 'Odell', 'Zay', 'Gus'],
    'Age': [61, 27, 31, 23, 35],
    'College': ['Miami (OH)', 'Louisville', 'LSU', 'Boston College', 'Rutgers']
}

df = pd.DataFrame(data)

# Displaying the data frame
print(df)

# Accessing columns
names = df['Name']
ages = df['Age']

# Filtering rows
```

```
young_people = df[df['Age'] < 30]

# Adding a new column
df['Gender'] = ['Female', 'Male', 'Male', 'Male', 'Female']

# Deleting a column
df.drop('College', axis=1, inplace=True)

# Summary statistics
summary_stats = df.describe()

# Grouping and aggregation
grouped_data = df.groupby('Gender')['Age'].mean()

# Sorting the data frame
sorted_df = df.sort_values(by='Age', ascending=False)

# Writing data to a CSV file
df.to_csv('output_data.csv', index=False)

# Reading data from a CSV file
csv_df = pd.read_csv('output_data.csv')
```

> Output:
>
> ```
>     Name  Age         College
> 0   John   61      Miami (OH)
> 1  Lamar   27      Louisville
> 2  Odell   31             LSU
> 3    Zay   23  Boston College
> 4    Gus   35         Rutgers
> ```

Data frames are an effective instrument for investigating, sanitizing, analyzing, and visualizing data in data science undertakings. They enable effortless manipulation of data, execution of diverse operations, and efficient handling of structured datasets.

4.10 Hashing

Hashing is a procedure that transforms data, such as text or numbers, into a string of characters with a fixed size. This resulting string is usually a distinct representation of the original input data. In Python, hash functions can be utilized for performing hashing operations.

Creating a Hash

```
# Importing the hashlib library for hash functions
import hashlib
```

```python
# Creating a simple hash
data = "Hello, world!"

# Creating a hash object using SHA-256 hash function
sha256_hash = hashlib.sha256()

# Updating the hash object with data
sha256_hash.update(data.encode())

# Getting the hexadecimal representation of the hash
hashed_data = sha256_hash.hexdigest()

# Displaying the hashed data
print("Original Data:", data)
print("Hashed Data:", hashed_data)

# Using hash() built-in function
value = "Hello"
hashed_value = hash(value)

# Displaying the hashed value
print("Hashed Value:", hashed_value)
```

> Output:
>
> ```
> Original Data: Hello, world!
> Hashed Data: 315f5bdb76d078c43b8ac0064e4a0164612b1fce77c869345bfc94c75894edd3
> Hashed Value: 3473782894911448306
> ```

Hashing is extensively utilized in the field of data science for a multitude of applications, including but not limited to, data indexing in databases, data integrity verification, and efficient data retrieval in hash tables. The primary objective of hash functions is to produce distinct hash values for different inputs, ensuring data consistency and enhancing security.

4.11 Trie

A trie is a type of tree structure that uses nodes to represent individual characters within a string. The nodes are linked together in a manner that enables efficient retrieval and searching of strings.

Creating a Trie

```python
# Define a TrieNode class
class TrieNode:
    def __init__(self):
        self.children = {}  # Dictionary to store child nodes
        self.is_end_of_word = False  # Marks the end of a word

# Define a Trie class
```

```
class Trie:
    def __init__(self):
        self.root = TrieNode()  # The root node of the trie

    # Insert a word into the trie
    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    # Search for a word in the trie
    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

# Create a trie
trie = Trie()

# Insert words into the trie
trie.insert("large")
trie.insert("language")
trie.insert("model")
trie.insert("text")

# Search for words in the trie
search_result_large = trie.search("large")
search_result_language = trie.search("language")
search_result_method = trie.search("method")

print("Search 'large':", search_result_apple)
print("Search 'language':", search_result_banana)
print("Search 'method':", search_result_orange)
```

> Output:
>
> ```
> Search 'large': True
> Search 'language': True
> Search 'method': False
> ```

Tries are extremely beneficial in the field of data science when it comes to tasks such as autocom-

plete, spell-checking, and the implementation of efficient search engines. They enable quick and effective operations involving strings.

5 Binary Search Trees (BST)

5.1 Binary Tree

A binary tree is a fundamental data structure used in computer science and consists of nodes where each node has at most two children, commonly referred to as the left child and the right child. Here are the key characteristics of a binary tree:

- Nodes: Each node in a binary tree contains some data and has at most two child nodes.
- Root: The topmost node in a binary tree is called the root node. It is the starting point for accessing the data in the tree.
- Edges: The connections between nodes are called edges. Each node (except the root) is connected by a directed edge from its parent node.
- Parent, Child, and Sibling: Each node (except the root) has one parent node and may have zero, one, or two child nodes. Nodes with the same parent are called siblings.
- Left and Right Child: Nodes in a binary tree can have at most two children: a left child and a right child. These children are positioned relative to their parent.
- Subtrees: A binary tree can be divided into smaller binary trees known as subtrees. Each subtree itself is a binary tree.
- Height: The height of a binary tree is the number of edges on the longest path from the root node to a leaf node. It represents the length of the longest downward path to a leaf from the root.
- Depth: The depth of a node is the number of edges from the root to that node. The root node has a depth of 0.
- Balanced vs. Unbalanced: A balanced binary tree is one where the left and right subtrees of every node differ in height by at most one. An unbalanced tree has a significant difference in heights between subtrees, which can lead to performance issues.
- Traversal: Binary trees can be traversed in different ways: Inorder (left-root-right), Preorder (root-left-right), Postorder (left-right-root), and Level Order (breadth-first).

Introduction to Binary Search Tree

Binary search trees are an important data structure for dynamic sets. The binary search tree property is a property imposed on every vertex of a binary search tree such that the following conditions hold.

- Accomplish many dynamic-set operations in $O(h)$ time, where h = height of tree.
- A binary tree is represented by a linked data structure in which
- $T.root$ points to the root of tree T.
- Each node contains the attributes
  - $\diamond$ $T.key$ (and possibly other satellite data).
  - $\diamond$ $T.left$ points to left child.

- ◇ $T.right$ points to right child.
- ◇ $T.p$ points to parent. $T.root.p = NIL$.

Stored keys must satisfy the binary-search-tree property.

- If y is located in the left subtree of x, then $y.key \leq x.key$.
- If y is located in the right subtree of x, then $y.key \geq x.key$.

Drawing a sample tree as follows will help with a visual representation.

The **binary search tree property** is a property imposed on every vertex of a binary search tree such that the following conditions hold. Let v_i and v_j be two vertices in the tree. For the sake of discussion, assume all key values are distinct (although, in general, this is not required).

If v_j is located in the left subtree of v_i, then $key[v_j] < key[v_i]$. On the other hand, if v_j is located in the right subtree of v_i, then $key[v_j] > key[v_i]$.

5.2 BST Traversal

The binary-search-tree property allows us to print keys in a binary search tree in order, recursively, using an algorithm called an inorder tree walk. Elements are printed in monotonically increasing order.

Any given binary tree can be traversed in a number of interesting ways. Tree traversal corresponds to visiting all of the vertices in a tree in some order.

Three approaches that yield interesting and useful results are:

- Inorder traversal
- Preorder traveral
- Postorder traversal

Each of these traversals can be performed in $\Theta(n)$ time where $n = |T|$, and we can represent these choices using the following single algorithms:

Algorithm 1 Tree Traversal

```
function TreeTraverse(root,type)
    if root = NIL then
        return
    if type ="preorder" then
        print(key[root])
    TreeTraverse(root.left,type)
    if type ="inorder" then
        print(key[root])
    TreeTraverse(root.right,type)
    if type ="postorder" then
        print(key[root])
```

Interpreting Algorithm 1, we note that *root* corresponds to a pointer to the root of a subtree, and *type* specifies the type of traversal from the set of values {preorder, inorder, postorder}. How **TreeTraverse**(*root*,"inorder") works:

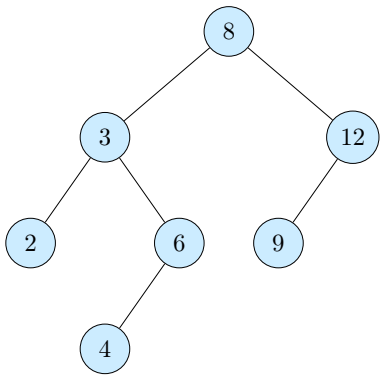


Figure 1: A binary search tree

- Check to make sure that x is not NIL.
- Recursively, print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

From this, we see that a preorder traversal involves "visiting" the root of the tree first (indicated by printing that nodes key value; however, any operation can be performed during a "visit"), then traversing the left and then right subtree. Consider Figure 1.

Example

In Figure‘1, a preorder traversal yields the sequence 8-3-2-6-4-12-9.

Before looking at the inorder traversal, let's consider postorder. In a postorder traversal, from a root node, we traverse the left subtree, then traverse the left subtree, and then visit the root. Again, using the tree in Figure 1, we see that postorder traversal yields the sequence 2-4-6-3-9-12-8.

Finally, let's consider the inorder traversal. For this traversal, we begin by traversing the left subtree from the root, then we visit the root node, and finally we traverse the right subtree. The reason we saved this for last is that, when applying an inorder traversal to a binary search tree, something interesting happens. The inorder traversal of this tree yields the sequence 2-3-4-6-8-9-12. In other words, an inorder traversal of any binary search tree will visit the vertices of the tree in non-decreasing order of the key values stored in the tree.

Correctness
Follows by induction directly from the binary-search-tree property.

Time
Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once. [Book has formal proof.]

The tree traversal corresponds to visiting all of the vertices in a tree in some order. Three approaches that yield interesting and useful results are:

- Inorder traversal
- Preorder traversal
- Postorder traversal

Each of these traversals can be performed in $\Theta(n)$ time where $n = |T|$, and we can represent these choices using Algorithm 1

5.3 Querying a Binary Search Tree

All dynamic-set search operations can be supported in $O(h)$ time. For a balanced binary tree the hight h has a running time of $h = \Theta(\log(n))$ (and for an average tree built by adding nodes in random order.) An unbalanced tree that resembles a linear chain of n nodes in the worst case is $h = \Theta(n)$.

Searching

Searching a binary search tree involves comparing the target key value with the key value of the root of a particular subtree. If the key value matches, then the search succeeds and returns a pointer to that vertex in the tree. If the key value does not match, the search proceeds left if the key value is less than the root's key value. It proceeds right otherwise. Should search reach a leaf of the tree without finding the target key value, search terminates by returning NIL.

Algorithm 2 Tree Search

```
function TreeSearch(x,k)
    if x == NIL or k == key[x] then
        return x
    if k < x.key then
        return TreeSearch(x.left, k)
    else
        return TreeSearch(x.right, k)
```

The initial call is to TreeSearch$(T.root, k)$.

Time

To analyze this algorithm, note that search proceeds in only one direction-down the tree. Therefore, the complexity of search is bound by the longest path from the root of the tree to a leaf, and this corresponds to the height of the tree h. Therefore, search complexity is $O(h)$. In the worst case, the tree could consist of a single linear chain, and h would equal n. Under these conditions, the complexity for searching becomes $O(n)$. On the other hand, if the tree is perfectly balanced, meaning all leaves are at the same depth (and there are roughly $n = 2$ leaves), then the complexity reduces to $O(\lg n)$.

[The text also gives an iterative version of TreeSearch, which is more efficient on most trees.

Minimum and Maximum

The binary-search-tree property guarantees that:

- the minimum key of a binary search tree is located at the leftmost node, and
- the maximum key of a binary search tree is located at the rightmost node.

Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time, where h is the height of the tree.

Finding the Minimum in the Tree

The structure of the tree is designed such that it is easy to find the minimum value stored in the tree. Since we know that all values less than some target must reside in a left subtree, all we need to do is traverse the tree from the root along left children until we find the first left NIL pointer. The value stored at that vertex is the minimum value in the tree.

Traverse the appropriate pointers ($left$) until NIL is reached.

Algorithm 3 Tree Minimum

```
function TreeMinimum(x)
    while x.left ≠ NIL do
        x = x.left
    return x
```

Finding the Maximum in the Tree

Finding the maximum value in the tree exactly parallels finding the minimum value. The only difference in the process is that the tree is traversed along right children.

Traverse the appropriate pointers ($left$ or $right$) until NIL is reached.

Algorithm 4 Tree Maximum

```
function TreeMaximum(x)
    while x.right ≠ NIL do
        x = x.right
    return x
```

Time

Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time, where h is the height of the tree.

Successor and Predecessor

Assuming that all keys are distinct, the successor of a node x is the node y such that $y.key$ is the smallest $key > x.key$. (We can find x's successor based entirely on the tree structure. No key comparisons are necessary.) If x has the largest key in the binary search tree, then we say that x's successor is NIL.

There are two cases:

- If node x has a non-empty right subtree, then x's successor is the minimum in x's right subtree.

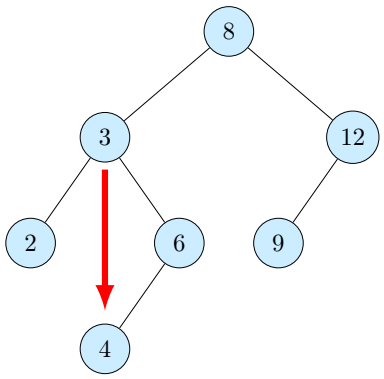


Figure 2: Case 1: Minimum value of subtree rooted at 3

- If node x has an empty right subtree, notice that:
  - ◇ As long as we move to the left up the tree (move up through right children), we're visiting smaller keys.
  - ◇ x's successor y is the node that x is the predecessor of (x is the maximum in y's left subtree).

Algorithm 5 Tree Successor

```
function TreeSuccessor(x)
    if x.right ≠ NIL then
        return TreeSuccessor(x.right)
    while y ≠ NIL and x == y.right do
        x = y
        y = y.p
    return y
```

TreePredecessor is symmetric to TreeSuccessor as shown in the following algorithm:

Algorithm 6 Tree Predecessor

```
function TreePredecessor(x)
    if x.left ≠ NIL then
        return TreePredecessor(x.left)
    while y ≠ NIL and x == y.left do
        x = y
        y = y.p
    return y
```

Example

As an example of Case 1, consider Figure 1 and suppose we want to find the successor to the vertex with key value 3. Since vertex 3 has a right subtree, we return the minimum of the subtree, and that has value of 4.

Now consider Case 2, and suppose we want to find the successor to vertex 6. This time, the vertex

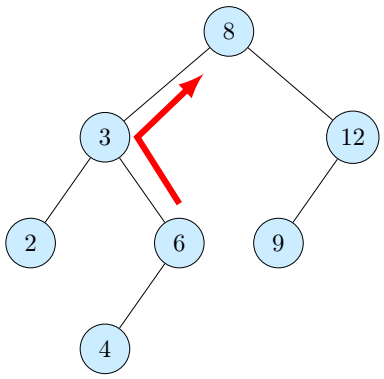


Figure 3: Case 2: Successor to vertex 6

does not have a right subtree, so we must travel up through the ancestors as long as we traverse a right-hand branch. As soon as we go up the first left-hand branch, that node is the successor. That node has value 8.

Notice that Case 1 corresponds to the minimum operation, so its complexity is the same as finding the minimum (i.e., $O(h)$). We also note that Case 2 only travels up the tree, so it too is bound by the height of the tree (i.e., $O(h)$).

Time

For both the TreeSucessor and TreePredecessor procedures, in both cases, we visit nodes on a path down the tree or up the tree. Thus, running time is $O(h)$, where h is the height of the tree.

5.4 Insertion and Deletion

Insertion and deletion allows the dynamic set represented by a binary search tree to change. The binary-search-tree property must hold after the change. Insertion is more straightforward than deletion.

Insertion

- To insert value into the binary search tree, the procedure is given node z, with $z.key = v, z.left = NIL$, and $z.right = NIL$.
- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - ◇ Pointer x: traces the downward path.
  - ◇ Pointer y: "trailing pointer" to keep track of parent of x.
- Traverse the tree downward by comparing the value of node at x with v, and move to the left or right child accordingly.
- When x is NIL, it is at the correct position for node z.
- Compare z's value with y's value, and insert z at either y's $left$ or $right$, appropriately.

Analysis of Insertion

Algorithm 7 Tree Insert

```
function TreeInsert(T, z)
    y = NIL
    x = T.root
    while x ≠ NIL do
        y = x
        if z.key < x.key then
            x = x.left
        elsex = x.right
    z.p = y
    if y == NIL then
        T.root = z                                   ▷ tree T was empty
    else if z.key < y.key then
        y.ledt = z
    else
        y.right = z
    return y
```

- The initialization takes $O(1)$ time to complete
- The while loop in lines 3-7 searches for place to insert z, maintaining parent y which takes $O(h)$ time.
- Lines 8-13 insert the value and takes $O(1)$

It should be noted that this is the same as TreeSearch. On a tree of height h, the procedure takes $O(h)$ time. TreeInsert can be used with InorderTreeWalk to sort a given set of numbers. (See Exercise 12.3-3 in [2])

Deletion

Deletion is the most complicate operation to be performed on a binary search tree.

[Deletion from a binary search tree changed in the third edition. In the first two editions, when the node z passed to TreeDelete had two children, z's successor y was the node actually removed, with y's contents copied into z. The problem with that approach is that if there are external pointers into the binary search tree, then a pointer to y from outside the binary search tree becomes stale. In the third edition, the node z passed to TreeDelete is always the node actually removed, so that all external pointers to nodes other than z remain valid.]

For deletion, three cases must be considered:

- The vertex to be deleted has no children. In this case, simply delete the vertex and set the pointer from the parent to the vertex to NIL.
- The vertex to be deleted has only one child. In this case, "splice out" the vertex by setting the pointer from the parent to the vertex to point to the vertex's child.
- The vertex to be deleted has two children. For this case, we first find the successor to the vertex being deleted, copy that vertex to the position being deleted, and then delete the successor vertex from the tree.

This case is a little tricky because the exact sequence of steps taken depends on whether y is z's right child. The code organizes the cases a bit differently. Since it will move subtrees around within the binary search tree, it uses a subroutine, TRANSPLAN, to replace one subtree as the child of its parent by another subtree.

Algorithm 8 Transplant

```
function TRANSPLANT(T, u, v)
    if u.p == NIL then
        T.root = z
    else if u == u.p.left then
        u.p.left = v
    else
        u.p.right = v
    if v ≠ NIL then
        v.p = u.p
```

TRANSPLANT(T, u, v) replaces the subtree rooted at u by the subtree rooted at v:

- Makes u's parent become v's parent (unless u is the root, in which case it makes the root).
- u's parent gets as either its left or right child, depending on whether u was a left or right child.
- Doesn't update $v.left$ or $v.right$, leaving that up to TRANSPLANT's caller.

TREEDELETE has four cases when deleting node z from binary search tree T:

- If z has no left child, replace z by its right child. The right child may or may not be NIL. (If z's right child is NIL, then this case handles the situation in which z has no children.)
- If z has just one child, and that child is its left child, then replace z by its left child.
- Otherwise, z has two children. Find z's successor y. y must lie in z's right subtree and have no left child.
  Goal is to replace z by y, splicing y out of its current location.
  - ◇ If y is z's right child, replace z by y and leave y's right child alone.
  - ◇ Otherwise, y lies within z's right subtree but is not the root of this subtree. Replace y by its own right child. Then replace z by y.

Note that the last three lines execute when z has two children, regardless of whether y is z's right child.

Example

Cases 1 and 2 are bound simply by the traversal process and so have complexity $O(h)$. We will show that Case 3 is also $O(h)$. But first, consider the following example. Suppose we wish to delete 3 from the following tree (Figure 4).

According to the algorithm, we find the successor (which is 4), copy it up to the deleted position, and then delete the successor vertex (Figure 5).

Algorithm 9 Transplant

```
function TRANSPLANT(T, u, v)
    if z.left == NIL then
        TRANSPLANT(T, z, z.right)                          ▷ z has no left child
    else if z.right == NIL then
        TRANSPLANT(T, z, z.leftt)                          ▷ z has just a left child
    else                                                   ▷ z has two children.
        TREEMINIMUM(z.right)                               ▷ y is z's successor
        if y.p ≠ z then
            ▷ y lies within z's right subtree but is not the root of this subtree.
            TRANSPLANT(T, y, y.right)
            y.right = z.right
            y.right.p = y
        ▷ Replace z by y.
        TRANSPLANT(T, z, y)
        y.left = z.left
        y.left.p = y
```

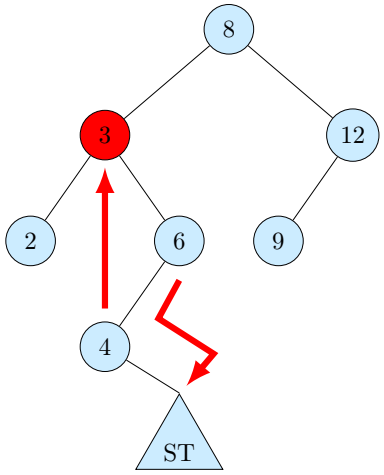


Figure 4: Deletion Case 3: Finding successor of 3

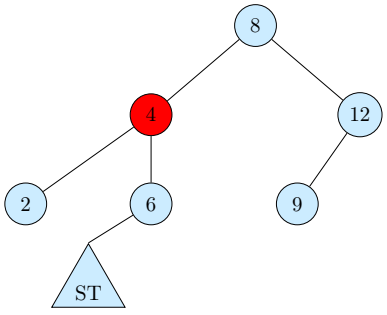


Figure 5: Deletion Case 3: After deletion node of 3

Time

$O(h)$, on a tree of height h. Everything is $O(1)$ except for the call to $Tree - Minimum$.

Minimizing running time

We've been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

- Problem: Worst case for binary search tree is $\Theta(n)$—no better than linked list.
- Solution: Guarantee small height (balanced tree)—$h = O(\lg(n))$.

In later chapters, by varying the properties of binary search trees, we will be able to analyze running time in terms of n.

- Method: Restructure the tree if necessary. Nothing special is required for querying, but there may be extra work when changing the structure of the tree (inserting or deleting).

Red-black trees are a special class of binary trees that avoids the worst-case behavior of $O(n)$ that we can see in "plain" binary search trees. Red-black trees are covered in detail in Chapter 13.

Binary Search Tree Conclusion

Binary search trees are viewed today as data structures that can support dynamic set operations, e.g., Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete. They can be used to build Dictionaries and Priority Queues. The basic operations take time proportional to the height of the tree – $O(h)$.

6 Algorithm Analysis

6.1 Basic Analysis

6.1.1 Summations

As we saw in the first unit, we can analyze several algorithms iteratively by considering the number of times through a particular loop. Recall that when analyzing insertion sort, we had several summations to consider. This, in fact, is commonplace when analyzing iterations. Unfortunately, handling such summations in raw form can be cumbersome, so much of algorithm analysis focuses on reducing summations to more manageable forms.
In the following we provide several reductions (without proof) to be used as needed in analysis.

Given a finite sequence of numbers $a_1, \ldots, a_n$, the sum $a_1 + + a_n$ can be written as $\sum_{i=1}^{n} a_i$.

Similarly, the sum over an infinite sequence $a_1 + a_2 + \ldots$ can be written as $\sum_{i=1}^{\infty} a_i$.

Given an infinite series, we say that the series diverges if a finite limit does not exist on the sum.

Otherwise, the series is said to *converge.*
An *absolutely convergent* series is a convergent series with the property that the sum of absolute values also converges, i.e., $\sum_{i=1}^{\infty} |a_i| < \infty$.
Given the definition of a summation and the basic properties of arithmetic, the following linear properties hold.

$$\sum_{i=1}^{n}(ca_i + b_i) = c\sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

$$\sum_{i=1}^{n} \Theta(f(i)) = \Theta\Big(\sum_{i=1}^{n} f(i)\Big), \text{ for asymptotic notation.}$$

There are several arithmetic series that will be of interest to us, including the following:

- The Arithmetic Series:
$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1) = \Theta(n^2)$$

- The Geometric Series:
$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

- The Harmonic Series:
$$\sum_{k=1}^{\infty} x^k = \frac{1}{1-x}, \text{ for } |x| < 1$$

- A Telescoping Series: In this series, each step adds a term subtracted from a previous step, namely

$$\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0$$
$$\sum_{k=0}^{m-1}(a_k - a_{k+1}) = a_0 - a_n$$

For example,

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n}$$

An interesting problem arises when faced with a product series (rather than a summation). What is interesting in this case is that the product can be converted to an equivalent sum and then "solved" in the normal way. Specifically,

$$\prod_{k=1}^{n} a_k = \lg^{-1}\left[\lg\left(\prod_{k=1}^{n} a_k\right)\right] = \lg^{-1}\left[\sum_{k=1}^{n} \lg a_k\right].$$

6.1.2 Bounding Summations

Given a summation, the goal in support of algorithm analysis is to find a "closed form solution" to that summation. Since much of algorithm analysis focuses on determining asymptotic bounds, often it is unnecessary to find an exact solution to the summation. Therefore, much analysis reduces to finding tight bounds on the summations.

6.1.3 Bounding Terms

In addition to determining a bound for the total sum, it may be convenient to consider bounds on specific terms of the sum. This can then be used to derive the final bound on the sum as a whole. Specifically, by our definitions of asymptotic notation, once we have bound the individual terms, we can safely say that the bound for the overall sum is driven by the bound of the largest term. In general,

$$\sum_{k=1}^{n} a_k \leq n \max_k\{a_k\}.$$

For example, suppose we want to bound $\sum_{k=1}^{n} k$.

Then we know that $\sum_{k=1}^{n} k < \sum_{k=1}^{n} n = n^2$.

Suppose the summation is bound by a geometric series. Then we can actually find a tighter bound than simply considering the largest term.

Specifically, suppose $\sum_{k=0}^{n} a_k$, $\frac{a_k+1}{a_k} \leq r, r < 1, k \geq 0$, and $a_k \leq a_0 r^k$.

Then we can derive a tighter bound by observing,

$$\sum_{k=0}^{n} a_k \leq \sum_{k=0}^{\infty} a_0 r^k = a_0 \sum_{k=0}^{\infty} r^k = a_0 \frac{1}{1-r}.$$

As a longer example, suppose we wish to bound $\sum_{k=1}^{\infty} \frac{k}{3^k}$.

Observe that, for $k = 1$, we have the sum reducing to $1/3$.

Now consider the ratio of consecutive terms as we did above.

$$\frac{(k+1)/3^{k+1}}{k/3^k} = \frac{1}{3} \cdot \frac{k+1}{k} \leq \frac{2}{3}.$$

Since the ratio is bound by 2/3, we can say that each term itself is bound from above by $\frac{1}{3} \cdot \left(\frac{2}{3}\right)^k$. Therefore,

$$\sum_{k=1}^{\infty} \frac{k}{3^k} \leq \sum_{k=1}^{\infty} \frac{1}{3} \left(\frac{2}{3}\right)^k = \frac{1}{3} \cdot \frac{1}{1-\frac{2}{3}} = 1.$$

6.1.4 Splitting Summations

When dealing with particularly difficult series, the idea of bounding the individual terms motivates yet another technique—splitting the summation into several shorter summations. Here the idea is to split a summation into two or more simpler summations, determine the appropriate bounds on each of those summations, and then derive the final bound. In using this approach, it is often sufficient to ignore a constant number of initial terms, thereby simplifying the summation further.

As an example, suppose we wish to find the bound on the normal Harmonic series: $H_n = \sum_{k=1}^{n} \frac{1}{k}$.

For this, we are going to split the summation into $[\lg n]$ pieces. We pick this rather odd splitting criterion because it will enable us to specify several simpler summations that are each bound above by 1.

In these simpler sums, each sum will begin with $1/2^i$ and go up to $1/2^{i+1}$.

Splitting in this fashion yields

$$\begin{aligned}
\sum_{k=1}^{n} \frac{1}{k} &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \\
&\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} \\
&\leq \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \\
&\leq \lg n + 1.
\end{aligned}$$

If we decompose each of the steps here, we see that the first line handles the splitting as described above.

The second line simplifies the inner sum by bounding each term from above by $1/2^i$. Because of this bound, we can replace each inner sum with 1 as an upper bound for that sum alone—this is what we designed by splitting the sum the way we did.

From here, the fourth line follows and completes the derivation.

Of interest with this derivation is the fact we proved the harmonic series was bound by $O(\lg n)$ where previously we claimed the series was bound by $O(\ln n)$. But recall we said that we could convert between logs using $\log_b a = \log_c a / \log_c b$. This is significant because $\log_c b$ is a constant. Therefore, all logs can be treated as if they are equivalent in an asymptotic sense.

6.2 Approximation with Integrals

A reasonable question to ask is why one cannot use integral calculus to bound summations. In fact, for certain summations, bounding with integrals is perfectly reasonable. Specifically, if our summation is over a monotonically increasing series of numbers, then we can approximate the bounds (upper and lower) as follows:

$$\int_{m-1}^{n} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx.$$

For example, suppose we have a simple monotonically increasing series. Then the lower bound is represented by the lower integral within the region in yellow in the following figure:

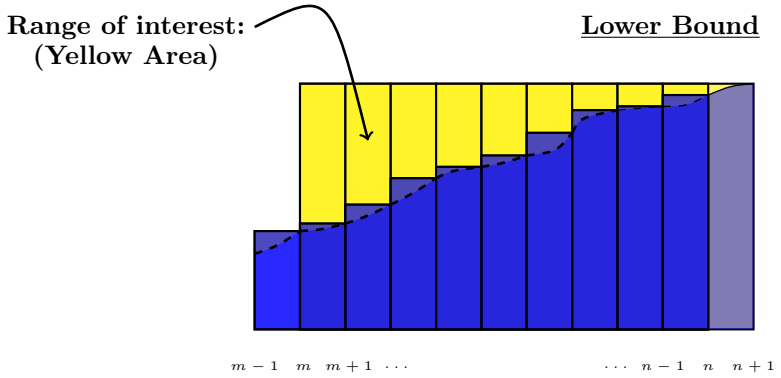


Figure 6: Lower bound example showing showing the region of interest in yellow.

In the same way, the upper bound is represented by upper integral within the region in yellow in the following figure:

Similarly, if our summation is over a monotonically decreasing series of numbers, then we can approximate the bounds (upper and lower) as follows:

$$\int_{m}^{n+1} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)dx.$$

As a detailed example, consider the Harmonic series once again.

Specifically, we wish to bound $H_n = \sum_{k=1}^{n} \frac{1}{k}$, and we note that this series is monotonically decreasing.

Therefore, we use the second set of integrals. Consider the upper bound first.
Then

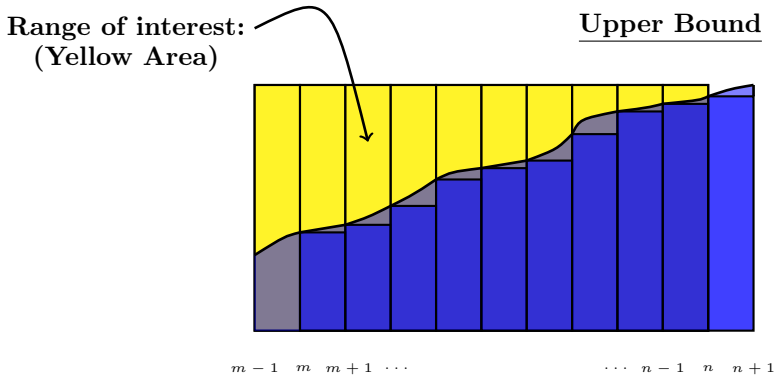


Figure 7: Upper bound example showing showing the region of interest in yellow.

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n} \frac{dx}{x}.$$

We solve the integral and find

$$\sum_{k=2}^{n} \frac{1}{k} \leq \ln x|_{1}^{n} = (\ln n - \ln 1) = \ln n.$$

Now we consider the lower bound. Here

$$\int_{2}^{n+1} \frac{dx}{x} \leq \sum_{k=2}^{n} \frac{1}{k}.$$

Once again, we solve the integral and find $\ln x|_{2}^{n+1} \leq \sum_{k=2}^{n} \frac{1}{k}$ so

$$(\ln(n+1) - \ln 2) = \ln\left(\frac{n+1}{2}\right) \leq \sum_{k=2}^{n} \frac{1}{k}.$$

Thus we have a tight bound of $H_n = \Theta(\ln n)$.

7 Insertion Sort

Problem Statement Input: A sequence of n numbers $< a_1, a_2, \ldots, a_n >$ Output: A permutation (reordering) $< a'_1, a'_2, \ldots, a'_n >$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

This is a good algorithm for sorting small number of elements.
For example:

- Start with an empty left hand.
- Add one card at a time in the correct position.
- Find the correct position for a card; compare it with each of the cards already in the hand from left to right.
- The cards are sorted at all times, the initial cards are not sorted as they are added one at a time.

Pseudocode:

1. The input parameter is an array $A[1..n]$ with a length of n
2. The ".." denotes a range within an array
3. When using an array $A[1..n]$ allocate the array to be one entry longer, i.e., $A[0..n]$
4. The array A is sorted in place.

Algorithm 10 Insertion Sort Algorithm

function INSERTIONSORT(A)	cost	time
for $j = 2$ To $A.length$ **do**	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted		
// sequence $A[1..j-1]$		
$i = j - 1$	c_3	$n-1$
while $i > 0$ and $A[i] > key$ **do**	c_4	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
$i = i - 1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] = key$	c_7	$n-1$

Correctness: Often a loop invariant is used to help understand why an algorithm gives the correct answer.

Loop Invariant: At the start of each iteration of the outer loop, the subarray $A[1 \ldots j-1]$ consists of the original elements from position $1, 2, \ldots, j-1$, but in sorted order.

To prove loop invariance is correct three things must be shown:

1. Initialization: Just before the first iteration, $j = 2$. The subarray $A[1, \cdots, j-1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.

2. Maintenance: To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving $A[j-j], A[j-2], A[j-3]$, and so on, by one position to the right until the proper position for key (which has the value that started out in$A[j]$) is found. At that point, the value of key is placed into this position.
3. Termination: The outer **for** loop ends when $j > n$, which occurs when $j = n+1$. Therefore, $j-1 = n$. Plugging n in for $j-1$ in the loop invariant, the subarray $A[1 \cdots n]$ consists of the elements originally in $A[1 \cdots n]$ but in sorted order. In other words, the entire array is sorted.

The loop invariant for Insertion-Sort starts each iteration of the "outer" for loop indexed by j. The sub array $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$ but in sorted order.

Analysis of Insertion Sort:

- Assume that the ith line takes c_i which is a constant. Note: line 3 is a comment so no time is taken.
- For $j = 2, 3, \ldots, n$, let t_j be the number of times the while loop test is executed for j.
- Note: that when a for or while loop exists in the usual way the test is executed one time more than the loop body.

The running time of the algorithm is equal to $\sum_{\text{over all statements}}$(cost of statement)$\times$(number of times statement is executed)

Let $T(n)$ = running time of Insertion-Sort
$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)$
The run time is dependent on the values of t_j.

Best Case: The array is already sorted.
$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$
$= (c_1 + c_2 + c_3 + c_4 + c_7)n \breve{} (c_2 + c_3 + c_4 + c_7)$

Worst Case: The array is sorted in reverse order.
$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$
$= \left(\frac{c_4+c_5+c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 \breve{} c_5 \breve{} c_6}{2} + c_7\right)n \breve{} (c_2 + c_3 + c_4 + c_7)$

Average Case: Typically the worse case is calculated instead of the average case due to the following three reasons:

1. The worst case run time provides an upper bound on the run time for an input.
2. The worst case often occurs, e.g., when searching the worst case often occurs when an item being searched for is not present.
3. The average case is not analyzed because it is often as bad as the worst case.

7.1 Loop Invariants

Loop invariants address an issue of how one goes about proving the correctness of an algorithm under the conditions where algorithms involves iteration. We begin by noting that the text shows three stages in proving correctness based on loop invariants:

- Initialization: It is true prior to the first iteration of the loop.
- Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination: When the loop terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

Using loop invariants is like mathematical induction:

- To prove that a property holds, you prove a base case and an inductive step.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely. We stop the "induction" when the loop terminates.
- We can show the three parts in any order.

To explain and justify these steps, we will look a bit at the theory behind loop invariants. We will also consider a simple example around which we will center our discussion, based on the following theorem.

Theorem: For any integer $n \geq 0$ and integer $m > 0$, there exist q and r such that $n = q \cdot m + r$ and
$0 \leq r < m$.

The way we will go about proving this theorem is by designing an algorithm to find q and r for any given n and m under the above conditions. We will then prove the algorithm is correct using loop invariants. This algorithm, by the way, is called the DIVISION algorithm and consists of the following:

Algorithm 11 Division

```
function DIVISION(int n, int m)
    int q, r;
    q ← 0
    r ← n
    while r ≥ m do
        q ← q + 1
        r ← r − m
    return q, r;
```

To prove this algorithm is correct (and thereby prove the theorem), we need to prove two things. First, we need to prove that the algorithm will terminate. Second, we need to prove that, when it terminates, it will terminate with the correct result. To accomplish the first (which corresponds to an important part of the Termination step in the text), we note that termination depends upon whether or not the while loop is an infinite loop. Notice that the initial value for r, which is the variable we test for termination, is n. Should $r < m$, we never enter the loop, and the algorithm terminates immediately. On the other hand, if $r \geq m$, we enter the loop. Now notice that the value of r is updated during each iteration of the loop with the statement $r \leftarrow r - m$. Recall that $m > 0$, so this means the value of r must reduce on each iteration, n, $n - m$, $n - 2m$, etc. Since r is strictly decreasing and $m > 0$, there must come a point where $r < m$, thus causing the loop to terminate. From this, we can conclude that for all values of n and m, the above algorithm must terminate.

For the second part of our proof, we will use the concept of a loop invariant. For this part, we need some additional background. First, for some terminology, notice that we are working with a while loop. In fact, what we are about to do works with any kind of loop, so we discuss the while loop without loss of generality. A while loop has the following general form:

while C **do**
    E;
end while

In this template, we note that C corresponds to the logical conditions that must be satisfied for the loop to be entered. This is called the "guard" of the loop. The E part of the loop corresponds to the statements that are executed inside the loop. The loop is said to terminate once the guard is no longer true.

Definition: Let S be some statement. Then S is an *invariant* of the loop

while C **do**
    E;
end while

when both S and C are true before any given iteration of the loop and S remains true after the iteration.

Theorem: Assume statement S is an invariant of the loop

while C **do**
    E;
end while

Assume also that S is true on the first entry to the loop. Then

1. If the loop terminates, S is true after the last iteration, and
2. If the loop does not terminate, S remains true after every iteration.

Proof: By assumption, S is true upon the first entry into the loop. Assume there exists an it-

eration of the loop such that, following that iteration, S is no longer true (i.e., S is false). Let k indicate the iteration where this occurs. Therefore, for iterations $i = 0, 1, \ldots, k-1$, S remains true. Furthermore, the guard C must still be true after the $(k-1)^{th}$ iteration for the k^{th} iteration to occur. By the definition of a loop invariant, given above, S must still be true after the next iteration, which is a contradiction. Therefore, S remains true if the loop terminations and remains true for all iterations of the loop, even if the loop does not terminate. Thus our theorem is proved.

We are now ready to use this theorem to finish our correctness proof of the DIVISION algorithm. For this part, we need to determine a loop invariant. Here we will use the condition specification of the algorithm to serve as our invariant, namely

$$S = (n = q \cdot m + r) \wedge (r \geq 0).$$

First, we can show that S is true upon the initial entry into the loop. This corresponds to the Initialization step described in the text. In this case, we have that $r \geq 0$ since $r = n$ and $n \geq 0$. We can show the second part of S to be true by substition, namely,

$$\begin{aligned} n &= q \cdot m + r \\ &= 0 \cdot m + r \\ &= n. \end{aligned}$$

Notice that if the loop is not executed, these conditions remain true, so now we need to consider the situation where the loop is executed. This corresponds to the Maintenance step in the text, and when combined with our termination conditions above, gives us the Termination step as well. We start by assuming S is true prior to a given iteration. (Note that this process is very similar to an inductive proof). Examining the body, E, of the loop, we find that q and r both change such that $q' = q + 1$ and $r' = r - m$. After this occurs, we update S such that

$$\begin{aligned} n &= q' \cdot m + r' \\ &= (q+1) \cdot m + (r - m) \\ &= q \cdot m + m + r - m \\ &= q \cdot m + r. \end{aligned}$$

We see that the relative values do not change, so S remains unchanged. We also find that $r' \geq 0$ because, to enter the loop $r \geq m$ and all we did was subtract m from r. Otherwise, this iteration would not have occurred. Thus we see that S is indeed a loop invariant (it is unchanged after every iteration). Therefore, since the algorithm has also been shown to terminate and return values of q and r such that S holds, we know the algorithm is correct.

7.1.1 Loop Invariants for InsertionSort

We often use a loop invariant to help us understand why an algorithm gives the correct answer. Here's the loop invariant for InsertionSort. At the start of each iteration of the "outer" for loop—the loop indexed by j—the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$ but in sorted order.

- Initialization: Just before the first iteration, $j = 2$. The subarray $A[1..j-1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.

- Maintenance: To be precise, we would need to state and prove a loop invariant for the "inner" while loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner while loop works by moving $A[j-1]$, $A[j-2,]$, $A[j-3]$ and so on, by one position to the right until the proper position for key (which has the value that started out in $A[j]$ is found. At that point, the value of key is placed into this position.
- Termination: The outer for loop ends when $j > n$, which occurs when $j = n+1$. Therefore, $j-1 = n$. Plugging n in for $j-1$ in the loop invariant, the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$ but in sorted order. In other words, the entire array is sorted.

Pseudocode conventions

Covering most, but not all, here. See book pages 20–22 for all conventions [2].

- Indentation indicates block structure. Saves space and writing time.
- Looping constructs are like in C, C++, Pascal, and Java. We assume that the loop variable in a for loop is still defined when the loop exits (unlike in Pascal).
- // indicates that the remainder of the line is a comment.
- Variables are local, unless otherwise specified.
- We often use objects, which have attributes. For an attribute *attr* of object x, we write $x.attr$. (This notation matches $x.attr$ in Java and is equivalent to $x \rightarrow attr$ in C++.) Attributes can cascade, so that if $x.y$ is an object and this object has attribute *attr*, then $x.y.attr$ indicates this object's attribute. That is, $x.y.attr$ is implicitly parenthesized as $(x.y).attr$.
- Objects are treated as references, like in Java. If x and y denote objects, then the assignment $y = x$ makes x and y reference the same object. It does not cause attributes of one object to be copied to another.
- Parameters are passed by value, as in Java and C (and the default mechanism in Pascal and C++). When an object is passed by value, it is actually a reference (or pointer) that is passed; changes to the reference itself are not seen by the caller, but changes to the object's attributes are.

8 Analyzing Algorithms

We want to predict the resources that the algorithm requires. Usually, running time. In order to predict resource requirements, we need a computational model.

8.1 Random-access machine (RAM) model

- Instructions are executed one after another. No concurrent operations.
- It's too tedious to define each of the instructions and their associated time costs.
- Instead, we recognize that we'll use instructions commonly found in real computers:
  - ◇ Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling). Also, shift left/shift right (good for multiplying/dividing by 2^k).
  - ◇ Data movement: load, store, copy.
  - ◇ Control: conditional/unconditional branch, subroutine call and return.

Each of these instructions takes a constant amount of time.

The RAM model uses integer and floating-point types.

- We don't worry about precision, although it is crucial in certain numerical applications.
- There is a limit on the word size: when working with inputs of size n, assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. ($\lg n$ is a very frequently used shorthand for $\log_2 n$.)
  - ◇ $c \geq 1 \Rightarrow$ we can hold the value of $n \Rightarrow$ we can index the individual elements.
  - ◇ c is a constant $\Rightarrow$ the word size cannot grow arbitrarily.

8.2 How do we analyze an algorithm's running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.

8.2.1 Input Size

Depends on the problem being studied.

- Usually, the number of items in the input. Like the size n of the array being sorted.
- But could be something else. If multiplying two integers, could be the total number of bits in the two integers.

- Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

8.2.2 Running Time

On a particular input, it is the number of primitive operations (steps) executed.

- Want to define steps to be machine-independent.
- Figure that each line of pseudocode requires a constant amount of time.
- One line may take a different amount of time than another, but each execution of line i takes the same amount of time c_i .
- This is assuming that the line consists only of primitive operations.
  - ◇ If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
  - ◇ If the line specifies operations other than primitive ones, then it might take more than constant time. Example: "sort the points by x-coordinate."

9 Asymptotic Notation

O-notation can be shown as follows:
$O(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.

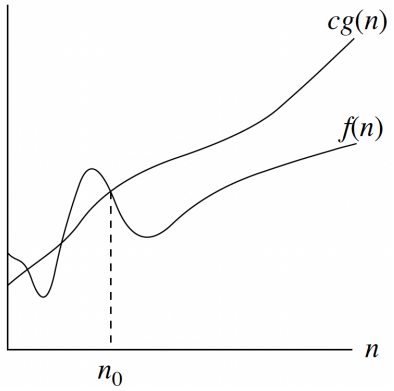


Figure 8: $f(n) = O(g(n))$ - O-notation upper bound example

O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. $g(n)$ is an asymptotic upper bound for $f(n)$.

Example:

$f(n) = 2n^2 = O(n^2)$, with $c = 1$ and $n_0 = 2$.

Other examples of functions $f(n)$ in $O(n^2)$

- $f(n) = n^2$
- $f(n) = n^2 + n$
- $f(n) = n^2 + 1000n$
- $f(n) = 1000n^2 + 1000n$

Ω-notation can be shown as follows:
$\Omega(g(n)) = \{f(n)$: there exists positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.

Ω-notation is a lower bound for a function $f(n)$ to within a constant factor. We write $f(n) = \Omega(g(n))$ if there exist positive constants n_0 and c such that at an to the right of n_0, the value of $f(n)$ always on or above $cg(n)$. $g(n)$ is an asymptotic lower bound for $f(n)$.

For example, $\sqrt{n} = \Omega(\lg n)$, with a constant $c = 1$ and $n_0 = 16$. Note the values of c and n_0 are examples.

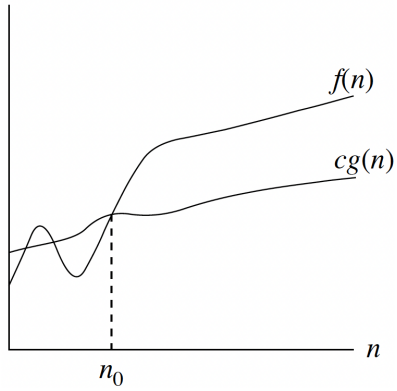


Figure 9: $f(n) = \Theta(g(n))$ - Ω-notation lower bound example

Θ-notation can be shown as follows:
$\Theta(g(n)) = \{f(n) : \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$.

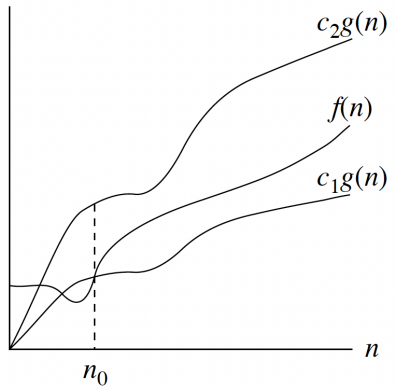


Figure 10: $f(n) = \Theta(g(n))$ - Θ-notation tighter bound example

Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1 and, c_2 such that at an to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. $g(n)$ is an asymptotic tight bound for $f(n)$.

For example, $n^2/2 - 2n = \Omega(n^2)$, with $c_1 = 1/4, c_2 = 1/2$ and $n_0 = 8$. Note the values of c_1, c_2 and n_0 are examples.

It should be noted that the leading constants and the lower order terms are not necessary in the analysis of the running time of an algorithm.

Theorem

For any two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. The leading constants and low-order terms do not matter.

9.1 Asymptotic Notation in Equations

When on the right hand side

$O(n^2)$ stands for some anonymous function in the set $O(n^2)$. $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means $2n^2 + 3n + 1 = 2n^2 + f(n)$ for some $f(n) \in \Theta(n)$. Specifically, $f(n) = 3n + 1$.

In the manner that the number of anonymous functions are interpreted will result in the number of time the asymptotic notation appears, for example:

$\sum_i^n = 1O(i)$	OK: 1 anonymous function
$O(1) + O(2) + \cdots + O(n)$	Not OK: n hidden constants $\implies$ no clean interpretation

When on the left hand side

No matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.

Interpret $2n^2 + \Theta(n) = \Theta(n^2)$ as meaning for all functions $f(n) \in \Theta(n)$, there exists a function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$.

Chaining the left hand side provides the following:

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Two equations can be considered as an interpretation for this example as follows:

1. There exists a function $f(n) \in \Theta(n)$ such that $2n^2 + 3n + 1 = 2n^2 + f(n)$
2. For all $g(n) \in \Theta(n)$ there exists $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$

9.2 Classes of Time Complexity

- Constant Time ($O(1)$): Regardless of the input size, the algorithm takes a constant amount of time to complete.
  - ◇ Accessing an element in an array by index.

- ◇ Inserting or deleting an element at the beginning of a linked list.

- Linear Time ($O(n)$): The runtime grows linearly with the size of the input.
  - ◇ Iterating through a list or array once.
  - ◇ Searching for an element in an unsorted list by traversing it.
- Logarithmic Time ($O(logn)$): As the input size grows, the runtime increases logarithmically.
  - ◇ Binary search in a sorted array.
  - ◇ Operations in a balanced binary search tree (BST).
- Quadratic Time ($O(n^2)$): The runtime grows quadratically with the input size.
  - ◇ Nested loops where each loop iterates through the input.
  - ◇ Some sorting algorithms like Bubble Sort or Selection Sort.
- Exponential Time ($O(2^n)$): The runtime doubles with each addition to the input size.
  - ◇ Algorithms involving recursive solutions that make repeated calls.
  - ◇ Solving the Traveling Salesman Problem using brute force.
- Factorial Time ($O(n!)$): The runtime grows factorially with the input size.
  - ◇ Permutation problems that explore all possible combinations, like the brute force solution for the Traveling Salesman Problem with no optimizations.

10 Recurrences

One of the most important tools for analyzing algorithms involves determining and then solving recurrences. A recurrence is an equation or inequality that describes a function in terms of its value on smaller and smaller inputs. Recurrences are typically used to characterize the performance of recursive functions/algorithms. As such, they are commonly applied in the analysis of "divide-and-conquer" algorithms. We considered the MergeSort algorithm in the Chapter 4 lecture slides as an example of a divide-and-conquer algorithm. In that chapter, we characterized MergeSort's complexity using the recurrence:

$$\begin{aligned} T(n) &= \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases} \\ &= \Theta(n \lg n) \end{aligned}$$

The question facing us is how we determined that the solution to this recurrence was $\Theta(n \lg n)$? More importantly, what tools are available to us to solve any recurrence?
In the following, we will discuss four different methods for solving recurrences:

- **The substitution method** - Here we will guess a bound and use mathematical induction to prove whether or not the guess is correct.
- **The iteration method** - For this method, we will convert the recurrence into a summation and bound the summation to solve.
- **Recursion trees** - In this case, we construct a tree showing the reduction of the input space and analyze the size of the tree.
- **The master method** - This approach provides a "cookbook recipe" for solving certain types of recurrences. Specifically, we will consider three cases of recurrences of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b > 1$, and $f(n)$ is some arbitrary function.

10.1 The Substitution Method

When applying the substitution method, we begin by guessing a bound that we believe reasonably represents the performance of the algorithm. Determining good guesses comes from experience after seeing a lot of recurrences with certain forms. Once the guess is made, mathematical induction is used to prove the bound.

As an example, we will attempt to find the upper bound of

$$T(n) = 2T(\lfloor n/2 \rfloor) + n.$$

We start by observing that this recurrence indicates the problem will be split in half upon each call, so we would expect the depth of the recursion to be no more than $O(\lg n)$. Since we are adding n with each recursion, we can guess that there is a multiplicative factor of n at each level. Thus, we will guess the bound is $O(n \lg n)$. In other words, we will prove via mathematical induction that $T(n) \leq cn \lg n$. To start our proof, we will assume the bound holds for $\lfloor n/2 \rfloor$. In other words, $T(n/2) \leq c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$. Given this, we will substitute back into the original recurrence (thus the name "substitution method") to yield the following:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n, \text{ for } c \geq 1. \end{aligned}$$

In this derivation, we substitute into the original recurrence in line 1. Line two drops the floors and combines the two $\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$ terms, requiring the inequality. Lines three and four are algebraic manipulations that permit the conclusion to be drawn in the last line. Note that this proof does not appear to follow the normal form for mathematical induction. In particular, note that there is no base case given. For most of the recurrences we consider, we will not worry about the base cases because they are trivial. More importantly, we note that asymptotic notation requires the relation $T(n) \leq cf(n)$ for some $n \geq n_0$. The proof can be simplified by removing the base cases for $n < n_0$, so the focus turns to the inductive step and determining the appropriate value for n_0.

As mentioned, one of the harder parts of the substitution method is determining a good guess. These guesses generally come from experience—from recognizing the form of the recurrence. If the form of the recurrence is not familiar, then a typical approach is to start with a loose bound and then attempt to tighten the bound. For the example, we just considered we know the bound is tight and can actually prove $T(n) = \Theta(n \lg n)$. We could have started with looser bounds (e.g., $O(n^2)$ and $\Omega(n)$), and then tightened the bounds to $O(n \lg n)$ and $\Omega(n \lg n)$ respectively.

When applying the substitution method, we must be careful. The inductive proof must prove the induc- tive hypothesis exactly. If the proof does not work, but intuition says that it should, then an approach to compensate is to subtract lower order terms in the guess. That said, one must be careful about guessing bounds that are too tight. Often one does not know the bounds are too tight until carefully considering the constants.

As an example, consider the recurrence $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$. n this case, we will guess that the bound is $O(n)$, so we need to prove that $T(n) \leq cn$. Applying substitution, we find:

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned}$$

At this point, one may think the proof is done. After all, applying the rules for asymptotic notation, we can drop lower order terms and declare $cn + 1$ to be $O(n)$. Right? Wrong. In this case, we observe that $cn + 1 > cn$, and this is significant. Specifically, it illustrates that we have not bound the recurrence from above by guessing $O(n)$, so something else must be done. We need a stronger inductive hypothesis. Well, our intuition continues to tell use that we do not want to go

to $O(n \lg n)$—that just seems to loose. So another approach is to attempt to subtract lower order terms. Specifically, this time we will guess that $T(n) \leq cn - b$ for some constant $b \geq 1$. Now when we apply the substitution method, we get

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/1 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b. \end{aligned}$$

which works for sufficiently large c and is what we wanted to show. Unfortunately, we are not done. We may still find ourselves in a situation where we have selected a bound that is too tight, but we miss a detail in the proof and seem to prove that the tight bound is correct. For example, let's attempt to prove that $T(n) = 2T(\lfloor n/2 \rfloor) + n = O(n)$. We already know that this is wrong, but what is wrong with the following proof?

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= (c+1)n \\ &= O(n). \end{aligned}$$

Well, just as $cn+1$ violated the upper bound requirement for our proof, so does $cn+n$. Specifically, $cn + n > cn$, and this violates the requirements for proving the inductive hypothesis.

Another technique that is sometimes used to assist with the substitution method is to change variables to represent something that is more manageable.

For example, let's attempt to solve the ugly recurrence $T(n) = 2T(\sqrt{n}) + \lg n$.
What are we going to do with the square root? Well, here we can convert to exponential form, and this might suggest a change in variables.

Specifically, if we rewrite the recurrence as $T(n) = 2T(n^{1/2}) + \lg n$, then we find something interesting happens if we let $m = \lg n$. Specifically, we can change the variables to be in terms of m and get $T(2^m) = 2T(2^{m/2}) + m$.
This doesn't look much better until we change the recurrence to use $S(m) = T(2^m)$. This allows us to substitute and rewrite the recurrence as $S(m) = 2S(m/2) + m$.
This is much easier to solve by substitution, and we will find $S(m) = O(m \lg m)$.
Now we have to convert back from $S(m)$ to $T(n)$, recalling that we assumed $m = \lg n$.
Doing this, we find $T(n) = O(\lg n \lg \lg n)$, and we are done.

10.1.1 Mathematical Induction

When determining (or guessing) a bound to a summation, one then needs to establish that the bound is reasonable and correct. One of the more widely accepted approaches to do this is to prove the bound holds via mathematical induction. An inductive proof consists of two parts: a base case and an inductive step. In the base case, the bound is proven for the "simplest" or "smallest" quantity or set of objects. When considering something like a recursive algorithm, the base case is directly analogous to the "bottom" of the recursion where no further recursive calls are required.

The inductive step, on the other hand, provides the mechanism by which the proof is extended to cover all possible cases. For the inductive step, the hypothesis to be proven is assumed to hold for some set, say of size n. Then the set is extended to consider the next logical size (e.g., $n+1$), and the hypothesis is proven to still hold. Since an arbitrary n is used, the extension can be argued to cover all possible sizes from that point forward.

To illustrate the process of completing a proof by induction, consider the following.
Suppose we wish to prove $\sum_{k=1}^{n} k = \frac{1}{2}n(n+1) = \Theta(n^2)$.

For this example, we will use a base case where $n = 1$.

Thus we want to consider $\sum_{k=1}^{1} k = 1$ relative to the previous hypothesis.

If we substitute $n = 1$ into the hypothesis, we find $\frac{1}{2}1(1+1) = \frac{1}{2}(2) = 1$, thus proving the base case.
Now we proceed to the inductive step and assume the hypothesis holds for $n = m$. The task now is to prove the hypothesis still holds for $n = m + 1$. We do this as follows:

$$\begin{aligned}
\sum_{k=1}^{m+1} &= \sum_{k=1}^{m} k + (m+1) \\
&= \frac{1}{2}m(m+1) + (m+1) \\
&= \left(\frac{1}{2}m + 1\right)(m+1) \\
&= \frac{1}{2}(m+2)(m+1) \\
&= \frac{1}{2}n(n+1).
\end{aligned}$$

Note that we can use mathematical induction directly on asymptotic bounds as well.

For example, suppose we wish to prove that $\sum_{k=0}^{n} 3^k = O(3^n)$.

Using the definition of big-Oh notation, this is equivalent to proving $\sum_{k=0}^{n} 3^k \leq c3^n$ for some constant c.

As before, we begin by specifying a base case.

In this case, we let the base case correspond to $n = 0$.

Now we consider the inductive step by assuming the hypothesis holds for $n = m$ and proving for $n = m + 1$.

$$\begin{aligned}\sum_{k=0}^{m+1} 3^k &= \sum_{k=0}^{m} 3^k + 3^{m+1} \\ &\leq c3^m + 3^{m+1} \\ &= \frac{c+3}{3c} c(3^{m+1}) \\ &= \left(\frac{1}{3} + \frac{1}{c}\right) c3^{m+1} \\ &\leq c'3^{m+1}\end{aligned}$$

Let's consider this derivation a little more carefully.

The first line occurs because we can decompose the summation into the original summation (corresponding to the inductive hypothesis) plus the extension.

The second line converts from the asymptotic form of the inductive hypothesis to the form based on the definition of big-Oh. After some algebraic manipulation, an identity is inserted to prepare for generation of a well-formed constant. We will discuss this notion of a well-formed constant below. Key to the idea is maintaining the requirement that the constant does not grow with increased input size.

In the third line, we derive this well-formed constant, c . We claim this constant is fine as long as $\frac{1}{3} + \frac{1}{c} \leq 1$, i.e., $c \geq \frac{3}{2}$. This is because we do not want our constant to exceed the value of the constant c since it must hold for all future steps in the series.

10.2 The Iteration Method

As discussed, being forced to guess a bound for the substitution method is problematic, especially when the level of experience is low. An alternative approach is the iteration method where the recurrence is expanded until a summation can be determined, then the summation is solved. The advantage to this approach is that guessing is not required. The disadvantage is that the algebra required can be messy. The idea is to expand the recurrence and express the resulting sum in terms dependent only on n and the initial conditions.

For example, we will solve the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + n$ using the iteration method. As a result, we get the following expansion:

$$\begin{aligned}T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/4 \rfloor/4)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/16 \rfloor/4))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor).\end{aligned}$$

With this, we see a decreasing geometric series. Specifically, we see the series as

$$T(n) \leq n + \frac{3}{4}n + \frac{9}{16}n + \frac{27}{64}n + \cdots + 3^{\log_4 n}\Theta(1).$$

[Note: Perhaps the most confusing part of this progression is the last term. We can see we are raising 3 to some power by examining the numerators in the coefficients. The reason for the logarithmic exponent is that we have a similar geometric progression in the denominator, growing as a factor of 4. This will cause the recurrence to "bottom out" on $\log_4 n$ steps.]
This can be rewritten as

$$T(n) \leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3})$$

Observe that we can rewrite $3^{\log_4 n}\Theta(1)$ as $\Theta(n^{\log_4 3})$ because of the identity $a^{\log_b n} = n^{\log_b a}$. Because of the geometric series, we can solve the summation as

$$T(n) = \frac{n}{1 - \frac{3}{4}} + \Theta(n^{\log_4 3}) = 4n + o(n)$$

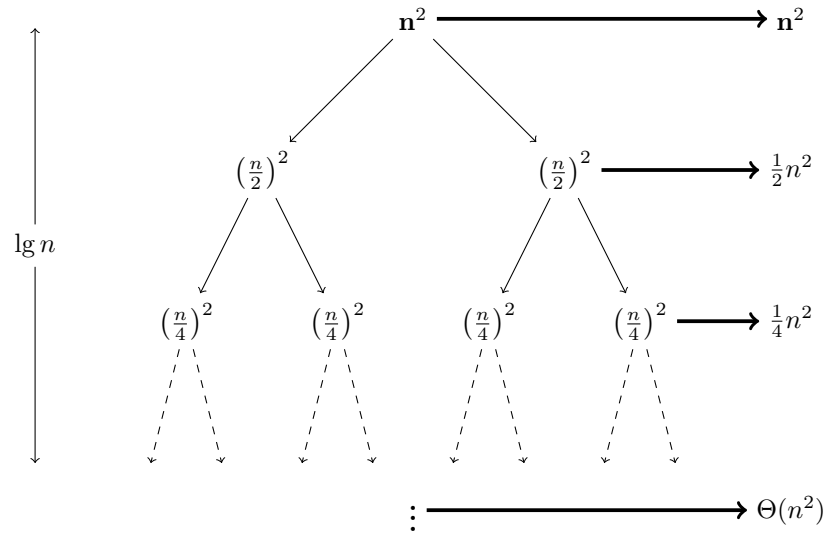
since $\log_4 3 < 1$. Thus we have that $T(n) = O(n)$.

A couple of tips to manage the iterations: First, to minimize confusion, focus on only two parameters—the sum of terms arising from the level of iteration and the number of times to recurse to hit the boundary. We saw this in the example above with the summation and the theta term respectively. As an alternative approach, sometimes the iteration method can be used to suggest a reasonable guess to start the substitution method.
So far, we have considered several recurrences with floors and ceilings. Often, it is fine to ignore the floors and ceilings and to focus on exact powers of numbers suggested by the denominator of the recursion. That said, there are times when such simplifications do not work, but they may still suggest a path forward.

10.3 Recursion Trees

Recursion trees provide a graphical method for handling the iteration method. The best way to explain their use is through examples.

Suppose we wish to solve the recurrence $T(n) = 2T(n/2) + n^2$. We can expand the recurrence tree to get the following:

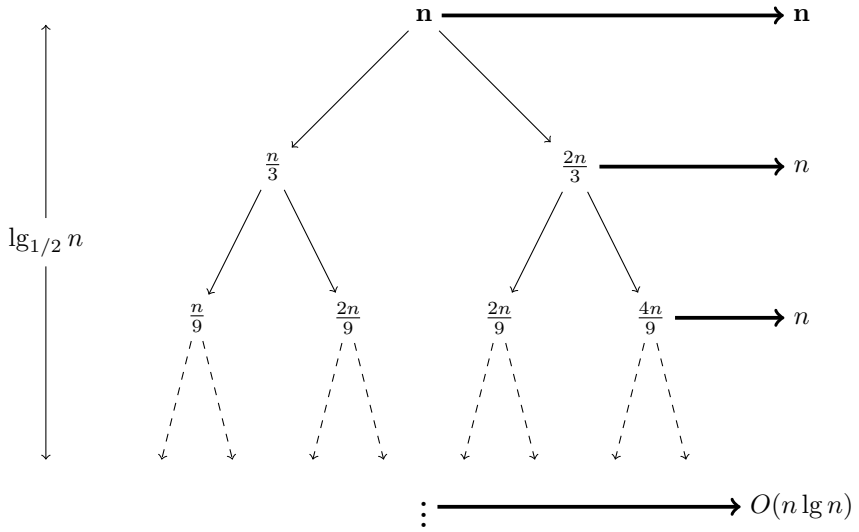


To read the figure above, we note that the depth of the tree is $\lg n$. This is equivalent to determining the depth of the recursion. Then consider the amount of work at each level of the tree. For this tree, it is interesting to note that a fraction of n^2 work is done at each level, and that decreases according to a geometric series. This can be converted to

$$\sum_{i=1}^{\lg n} \left(\frac{1}{2}\right)^i n^2 = n^2 \sum_{i=1}^{\lg n} \left(\frac{1}{2}\right)^i \leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2(n^2) = O(n^2).$$

Let's consider another example.

Suppose we wish to solve the recurrence $T(n) = T(n/3) + T(2n/3) + n$. The recurrence tree is as follows:



This time, we note that the tree is not balanced, but the longest branch is $O(\log_{3/2} n)$ deep. We also note that, through the depth of the shortest path, each level has $O(n)$work. Since the amount

of work at lower levels is less, and the depth of the shortest path is $O(\log_3 n)$, we have that the overall complexity is $O(n \lg n)$, remembering once again that the base of the log does not matter.

10.4 The Master Method

If we consider a particular form of recurrence, namely recurrences that fit the form for $T(n) = aT(n/b) + f(n)$ for $a \geq 1, b > 1$, and some asymptotically positive function $f(n)$, we can apply a "cookbook" method to solve them. We will provide the method here without proving the correspond theorem; however, the student is encouraged to examine the theorem and associated proof on their own.

To apply the method, we observe that this recurrence fits the general form of a "divide-and-conquer" recurrence where the input set of size n is divided into a subproblems, each of size n/b. The cost to combine is reflected in the function $f(n)$.

Theorem:

Let $a \geq 1$ and $b > 1$ be constants.
Let $f(n)$ be an asymptotically positive function.
Let $T(n)$ be defined on the non-negative integers by the recurrence $T(n) = aT(n/b) + f(n)$.
Interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ as appropriate. Then $T(n)$ can be bound asymptotically according to three cases:

- If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
- If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Intuitively, the solution to a divide-and-conquer recurrence is based on the larger of two functions – $f(n)$ and $n^{\log_b a}$. Whichever dominates will determine the bound.
Case 1 focuses on when $f(n)$ is smaller, in which case $n^{\log_b a}$ dominates and forms the bound.

Case 2 focuses on when $f(n) = n^{\log_b a}$ (or are at least the same order of magnitude).

Case 3 focuses on when $f(n)$ is larger, in which case $f(n)$ dominates and forms the bound. Note that we use "smaller" and "larger" in terms of polynomial differences between the functions. Note also that Case 3 considers a situation where $f(n)$ might fall in the "netherworld" between the cases where the master method cannot be applied.

To illustrate the use of the master method, consider the following examples:

Example 1: $T(n) = 9T(n/3) + n$
First, we note that $a = 9$ and $b = 3$. We also note that $f(n) = n$. We derive $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$.
Comparing with $f(n)$, we see that $f(n)$ is smaller, so this problem corresponds to Case 1.
Formally, we can let $\epsilon = 1$ so that $f(n) = O(n^{\log_3 9 - 1}) = O(n)$.
Because Case 1 applies, we have $T(n) = \Theta(n^2)$.

Example 2: $T(n) = T(2n/3) + 1$

First we note that $a = 1$ and $b = 3/2$.
We also note that $f(n) = 1$.
We derive $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.
Here we see that $f(n) = \Theta(n^{\log_b a})$, so Case 2 applies.
Thus $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$.

Example 3: $T(n) = 3T(n/4) + n \lg n$

First we note that $a = 3$ and $b = 4$. We also note that $f(n) = n \lg n$.
We derive $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$.
Letting $\epsilon = 0.2$, we have that $f(n) = \Omega(n^{\log_4 3 + \epsilon}) = \Omega(n)$.
Case 3 appears to apply, but we have to make sure we have not fallen into the netherworld.
Specifically, we need to check to see if $af(n/b) \leq cf(n)$ for some $c < 1$. Let $c = 3/4$.
Then we have $3(n/4)\lg(n/4) \leq \frac{3}{4} n \lg n$, and this is certainly true.
So Case 3 does apply, and we have $T(n) = \Theta(n \lg n)$.

10.5 Master Method Special Case

As a final comment, we introduce a "special case" of the master method that can come in handy. As we present this, we also note that the special case does not yield a theta bound while the full master method does.

Suppose we have a recurrence of the form $T(n) = a_1 T(n/b_1) + a_2 T(n/b_2) + \cdots + a_m T(n/b_m) + 1$.
Now suppose we define some $S = \sum_i (a_i/b_i)$. Then we have three cases as before:

1. If $S < 1$, then $T(n) = O(n)$
2. If $S = 1$, then $T(n) = O(n \lg n)$
3. If $S > 1$, then $T(n) = O(n^{\lg max_i a_i})$

As a final note, the coefficient on the final function n **must** be 1 for the case to hold.

11 Algorithm Examples

11.1 Bubble Sort

Bubble sort works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name because smaller elements bubble to the top of the list [2, 6, 3].

Because each of the n elements in an array might be compared with each other element in the array (in the worst case), the worst-case running time of bubble sort is $O(n^2)$. More specifically, the first pass requires $n-1$ comparisons, the second pass $n-2$, and so on, down to just 1 comparison. The sum of these successive terms (which forms an arithmetic series) is $\frac{n(n-1)}{2}$, which simplifies to $O(n^2)$ in terms of time complexity.

Algorithm 12 Bubble Sort

```
function BubbleSort(A)
    for i = 1 to length(A) − 1 do
        for j = 1 to length(A) − i do
            if A[j] > A[j + 1] then
                Swap A[j] and A[j + 1]
    return A
```

The Algorithm 12 defines a function BubbleSort that takes an array A and sorts it in ascending order using the bubble sort algorithm.

11.2 Merge Sort

Merge sort recursively divides the array into two halves, sorts each half, and then merges the two sorted halves together. The divide step computes the midpoint of the array, the conquer step recursively sorts the subarrays, and the combine step merges the sorted arrays into a single sorted array [2, 6, 3].

Algorithm 13's running time comes from the fact that each division approximately halves the number of elements in the subarrays (logarithmic growth in the number of divisions), and in each merge operation, all elements in the subarrays are processed (linear work). Therefore, the overall running time of the merge sort algorithm is $O(n \log n)$.

11.3 Binary Search

The process of binary search begins by comparing the middle element of a sorted array to the target value. If the target value matches the middle element, its position in the array is returned. If the target value is less than the middle element, the search continues in the lower half of the array. If the target value is greater, the search continues in the upper half of the array. This process repeats, each time cutting the array in half, hence the name “binary" search, until the target value is found or the subarray has zero length [2, 6, 3].

The running time of the binary search Algorithm 14 is $O(\log n)$ because each step cuts the list in half (hence the “log" term). The base of the logarithm is 2 because the list size is divided by two at each step.

Algorithm 13 Merge Sort Algorithm

```
function MergeSort(A, p, r)
    if p < r then
        q ← ⌊(p + r)/2⌋
        MergeSort(A, p, q)
        MergeSort(A, q + 1, r)
        Merge(A, p, q, r)
function Merge(A, p, q, r)
    n_1 ← q − p + 1
    n_2 ← r − q
    Let L[1 . . . n_1 + 1] and R[1 . . . n_2 + 1] be new arrays
    for i ← 1 to n_1 do
        L[i] ← A[p + i − 1]
    for j ← 1 to n_2 do
        R[j] ← A[q + j]
    L[n_1 + 1] ← ∞
    R[n_2 + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r do
        if L[i] ≤ R[j] then
            A[k] ← L[i]
            i ← i + 1
        else
            A[k] ← R[j]
            j ← j + 1
```

Algorithm 14 Binary Search Algorithm

```
function BinarySearch(A, x)
    low ← 1
    high ← length(A)
    while low ≤ high do
        mid ← ⌊(low + high)/2⌋
        if A[mid] == x then
            return mid
        else if A[mid] < x then
            low ← mid + 1
        else
            high ← mid − 1
    return NOT FOUND
```

11.4 Linear Search

In linear search, the algorithm iterates over all elements of the list, comparing each element with the target value. If the target value matches an element, the index of that element is returned. If the target value is not found by the end of the list, the search returns an indication that the target is not present in the list [2, 6, 3].

The running time of linear search Algorithm 15is $O(n)$, because in the worst case, it checks every element in the list exactly once. Thus, the time complexity is linearly dependent on the number of elements in the list.

Algorithm 15 Linear Search Algorithm

```
function LinearSearch(A, x)
    for i ← 1 to length(A) do
        if A[i] == x then
            return i
    return NOT FOUND
```

11.5 Selection Sort

The selection sort algorithm maintains two subarrays within the given array:

1. The subarray which is already sorted. The remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element from the unsorted subarray is picked and moved to the sorted subarray. This process continues moving the boundary of the unsorted subarray one element to the right each time [2, 6, 3].

However, it is important to note that the running time of selection sort Algorithm 16 is $O(n^2)$ in the average and worst cases. This is because, for each element in the array, the algorithm must search through the remaining part of the array to find the minimum element, making the number of comparisons proportional to $n(n-1)/2$.

Algorithm 16 Selection Sort Algorithm

```
function SelectionSort(A)
    n ← length(A)
    for i ← 1 to n − 1 do
        min_index ← i
        for j ← i + 1 to n do
            if A[j] < A[min_index] then
                min_index ← j
        Swap(A[i], A[min_index])
procedure Swap(a, b)
    temp ← a
    a ← b
    b ← temp
```

11.6 Dijkstra's Algorithm

Dijkstra's Algorithm works by iteratively finding the vertex with the minimum distance from the source, updating the path lengths for its adjacent vertices, and marking it as visited. The algorithm

keeps track of the currently known shortest distance from each vertex to the source and improves these values step-by-step [2, 6, 3].

The Dijkstra's Algorithm 17 has a running time complexity of $O(V^2)$ when implemented using a simple array, where V is the number of vertices. This complexity arises because, in each iteration, it needs to find the vertex with the minimum distance from the set of vertices that have not yet been included in the shortest path tree. This search operation takes $O(V)$ time, and it needs to be performed V times.

Algorithm 17 Dijkstra's Algorithm

```
function DIJKSTRA(G, s)
    dist[] ← array[1 ... |V|]          ▷ The output array. dist[i] will hold the shortest distance from s to i
    visited[] ← array[1 ... |V|]
    initialize dist[] to ∞, visited[] to false
    dist[s] ← 0
    for count ← 1 to |V| − 1 do
        u ← MINDISTANCE(dist[], visited[])
        visited[u] ← true
        for v ← 1 to |V| do
            if ¬visited[v] and G[u][v] and dist[u] ≠ ∞ and dist[u] + G[u][v] < dist[v] then
                dist[v] ← dist[u] + G[u][v]
function MINDISTANCE(dist[], visited[])
    min ← ∞
    min_index ← −1
    for v ← 1 to |V| do
        if visited[v] = false and dist[v] ≤ min then
            min ← dist[v], min_index ← v
    return min_index
```

11.7 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm iteratively revises path lengths between all pairs of vertices (i, j) including those that pass through intermediate vertices. For each pair of vertices, the algorithm considers whether a shorter path between them exists through an intermediate vertex k. This update is done using the recurrence relation:

$$d[i][j] = min(d[i][j], d[i][k] + d[k][j]) \tag{1}$$

where $d[i][j]$ is the shortest distance from vertex i to vertex j [2, 6, 3].

The running time of the Floyd-Warshall Algorithm 18 is $O(V^3)$ because there are three nested loops, each iterating through the vertices in the graph.

11.8 Quick Sort

Here's how Quick Sort works:

1. Pivot Selection: Select an element from the array as the pivot. The choice of pivot affects the performance of the sorting process.
2. Partitioning: Rearrange the array so that elements less than the pivot are on the left, the pivot is in the middle, and elements greater than the pivot are on the right.

Algorithm 18 Floyd-Warshall Algorithm for All-Pairs Shortest Paths

```
function FloydWarshall(W)
    n ← length(W)
    D^(0) ← W                                   ▷ Initial distance is the weight matrix
    for k ← 1 to n do
        Let D^(k) ← array[1...n][1...n]
        for i ← 1 to n do
            for j ← 1 to n do
                D^(k)[i][j] ← min(D^(k−1)[i][j], D^(k−1)[i][k] + D^(k−1)[k][j])
        D^(0) ← D^(k)                           ▷ Update distance matrix
    return D^(n)
```

3. Recursively Apply: Apply the same steps recursively to the sub-arrays formed by partitioning.

The average-case running time of Quick Sort Algorith 19 is $O(n \log n)$, which occurs when the partitions are balanced. This results because the array is divided in half with each pass (logarithmic number of passes), and each pass requires linear time n. However, in the worst case, if the partitioning is consistently unbalanced, the running time can degrade to $O(n^2)$. This worst-case scenario happens when the smallest or largest element is always picked as the pivot.

Algorithm 19 Quick Sort Algorithm

```
function QuickSort(A, p, r)
    if p < r then
        q ← Partition(A, p, r)
        QuickSort(A, p, q − 1)
        QuickSort(A, q + 1, r)
function Partition(A, p, r)
    x ← A[r]
    i ← p − 1
    for j ← p to r − 1 do
        if A[j] ≤ x then
            i ← i + 1
            Swap(A[i], A[j])
    Swap(A[i + 1], A[r])
    return i + 1
procedure Swap(a, b)
    temp ← a
    a ← b
    b ← temp
```

11.9 Knapsack Problem

There are several variations of the Knapsack Problem, but one of the most common is the 0/1 Knapsack Problem, where each item can either be taken or not taken (you cannot take a fractional part of an item) [4, 2, 3].

In Algorithm 20 the Dynamic Programming (DP) approach is used. For the 0/1 Knapsack Problem, a dynamic programming solution is often used. This approach involves creating a table where each entry $dp[i][w]$ represents the maximum value that can be achieved with the first i items and a maximum weight of w.

The algorithm's running time in this approach is $O(nW)$, where n is the number of items and W is the total capacity of the knapsack. This is considered pseudo-polynomial time, as it depends linearly on the number of items and the capacity of the knapsack, but exponentially on the log of the capacities (due to the capacity being part of the input size).

Algorithm 20 0/1 Knapsack Problem using Dynamic Programming

```
function KnapsackDP(v[], w[], W)
    n ← length(v)                                   ▷ Number of items
    dp ← array[0...n][0...W]                        ▷ DP table
    for i ← 0 to n do
        for j ← 0 to W do
            if i == 0 or j == 0 then
                dp[i][j] ← 0
            else if w[i − 1] ≤ j then
                dp[i][j] ← max(dp[i − 1][j], v[i − 1] + dp[i − 1][j − w[i − 1]])
            else
                dp[i][j] ← dp[i − 1][j]
    return dp[n][W]
```

11.10 Traveling Salesman Problem

The TSP is known for being NP-hard, meaning there is no known polynomial-time algorithm to solve it exactly for all general cases. Various algorithms exist that approach the problem with different strategies, ranging from brute force, to dynamic programming, to approximation and heuristic algorithms like genetic algorithms or simulated annealing [5, 2, 3].

1. Brute Force Approach: The brute force Algorithm 21 for the TSP checks every possible permutation of cities to determine the shortest possible route. Since there are $(n-1)!$ permutations (excluding the starting point if it is fixed), the time complexity of this approach is $O(n!)$. This exponential growth makes the brute force approach feasible only for very small datasets [1].
2. Dynamic Programming Approach (Held-Karp Algorithm 22): Solves the problem using $O(n^2 2^n)$ time and $O(n2^n)$ space complexities. This approach is more feasible than brute force for small to medium-sized problems.
3. Approximation Algorithms: For example, the Christofides algorithm guarantees a solution within 1.5 times the optimal length for problems that satisfy the triangle inequality. Its complexity is polynomial.

Algorithm 21 Brute Force Approach to the Traveling Salesman Problem

```
function BruteForceTSP(cities)
    n ← length(cities)
    min_path_length ← ∞
    best_path ← None
    for all perm ∈ Permutations(cities) do
        current_length ← 0
        for i ← 1 to n − 1 do
            current_length ← current_length + Distance(perm[i], perm[i + 1])
        current_length ← current_length + Distance(perm[n], perm[1])        ▷ Complete the cycle
        if current_length < min_path_length then
            min_path_length ← current_length
            best_path ← perm
    return best_path
```

Algorithm 22 Held-Karp Algorithm for Traveling Salesman Problem

```
function HeldKarp(D)
    n ← length(D)                                                    ▷ Number of cities
    C ← map()                                                        ▷ Cost map
    for all subsets S ⊆ {2, . . . , n} of size ≥ 2 do
        for i ∈ S do
            S′ ← S − {i}
            if S′ = ∅ then
                C[(S′, i)] ← D[1][i]
            else
                C[(S′, i)] ← min_{k∈S′}{C[(S′ − {k}, k)] + D[k][i]}
    return min_{i≠1}{C[({2, . . . , n}, i)] + D[i][1]}
```

12 Module Questions

13 Kaggle Datasets

Here are some Kaggle datasets that are particularly well-suited for the mentioned algorithms:

14 Module Questions and Answers

References

[1] David L. Applegate et al. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

[2] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. MIT Press, 2009.

[3] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th. MIT Press, 2022.

[4] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Pearson, 2005.

[5] E. L. Lawler et al. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, 1985.

[6] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th. Addison-Wesley Professional, 2011.