CALIFORNIA STATE UNIVERSITY
FULLERTON

**CPSC 535: Advanced Algorithms (Spring 2024)**

**Department of Computer Science, College of Engineering and Computer Science**

*Instructor: Dr. Shah, Adjunct Faculty, Computer Science https://www.hassanonline.us/*

*Wi-Fi Connectivity Product Management, Qualcomm Inc., San Diego.*

**Lecture # 2 Notes**

**Topic: Algorithm Efficiency Analysis:**

Importance:

- Determines how fast or resource-efficient a given algorithm is.
- Essential for ensuring user-friendly software experiences.

1. RAM Model:

Concept:

- A theoretical model of computation where each simple operation takes exactly one time step.
- Real-world Example:
  - Cashier at a Supermarket:
    - Scanning items is equivalent to performing operations.
    - Each item is processed individually and in sequence.

Python Code:

```
def sum_of_elements(elements):
    total = 0
    for element in elements:
        total += element
    return total
```

3. Step Counting:

Concept:

- Counting the number of operations or steps an algorithm takes.

Real-world Example:

- Page Counting:
  - If a book has 300 pages, you'd count 300 times.

Python Code:

```
def count_elements(elements):
    count = 0
    for _ in elements:
        count += 1
    return count
```

4. Asymptotic Analysis & Notations:

- Big O (Upper Bound):
  - Describes the upper limit of time an algorithm can take.
- Example:
  - Searching for a Friend in a Stadium:
    - Worst-case scenario, they're the last person you find.

Python Code:

```
def find_person(stadium, friend):
    for person in stadium:
        if person == friend:
            return True
    return False
```

- Big Ω (Lower Bound):
  - Describes the best amount of time an algorithm can take.

- Tight Bound (Θ):
  - Represents both the upper and lower bound.

- Example:
  - Binary Searching a Phone Book:
    - More efficient than checking each name individually.

Python Code:

```
def binary_search(array, x):
    low = 0
    high = len(array) - 1
    mid = 0
```

```
    while low <= high:
        mid = (high + low) // 2
        if array[mid] < x:
            low = mid + 1
        elif array[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1
```

5. Computational Problems: Linear Sorting:

Concept:

- Sorting elements in linear time.

Example: Organizing Colored Balls:

- Arrange balls from lightest to darkest shade.

Python Code:

```
def counting_sort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)

    for a in arr:
        count[a] += 1

    sorted_arr = []
    for i, c in enumerate(count):
        sorted_arr.extend([i] * c)
    return sorted_arr
```

6. Fundamental Data Structures:

- Arrays:
  - Continuous memory locations used to store items.
- Linked Lists:
  - Elements with data and reference to the next element.

- Stack:
  - Last-in-first-out (LIFO) structure.
- Queue:
  - First-in-first-out (FIFO) structure.

Closing Remarks:

Connecting theory to real-world examples helps in better comprehension and application. Understanding the efficiency and structure of algorithms will be beneficial for designing efficient software solutions. Enjoy Coding and Practicing the Sorting Algorithms

---

**Sorting Algorithm Real-World Examples:**

1. Bubble Sort

Description:
Bubble Sort is a simple sorting algorithm that works by repeatedly stepping through the list, 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, indicating that the list is sorted.

Example:
Consider a row of dancers who are all out of order, and they need to stand by increasing height order. Each dancer can only swap places with the dancer next to them. They repeatedly swap until everyone is in the correct position.

Code:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr

arr = list(map(int, input("Enter numbers separated by spaces: ").split()))
print("Sorted Array:", bubble_sort(arr))
```

2. Insertion Sort

Description:
Insertion sort builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

Example:
Think of playing cards. When you're dealt cards during a game, you often arrange them in your hand from lowest to highest. You take one card at a time and insert it into its correct position.

Code:
```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >=0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
    return arr

arr = list(map(int, input("Enter numbers separated by spaces: ").split()))
print("Sorted Array:", insertion_sort(arr))
```

3. Merge Sort

Description:
Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, sorts them, and then merges the two sorted halves.

Example:
Imagine you have two stacks of sorted papers but need them combined into one. You'd look at the top of each stack, take the smaller of the two papers, and move it to a new stack, repeating until one stack is empty and then taking from the other.

Code:
```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
```

```
        if L[i] < R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1
    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1
return arr

arr = list(map(int, input("Enter numbers separated by spaces: ").split()))
print("Sorted Array:", merge_sort(arr))
```

4. Heap Sort

Description:
Heap Sort is a comparison-based sorting technique based on the Binary Heap data structure. It divides its input into a sorted and an unsorted region and iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.

Example:
Imagine a classroom where students are called one by one according to their height to stand in front of the class. The tallest stands first, followed by the second tallest, and so on, ensuring that the line remains sorted.

Code:

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

```
def heap_sort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

arr = list(map(int, input("Enter numbers separated by spaces: ").split()))
print("Sorted Array:", heap_sort(arr))
```

5. Quick Sort

Description:
QuickSort is a Divide and Conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

Example:
Imagine you're a librarian who wants to sort books by weight, and you have a magic scale that can quickly divide the books into heavier and lighter than a given book (the pivot). By recursively applying this process, the books get sorted.

Code:

```
def partition(arr, low, high):
    i = low-1
    pivot = arr[high]
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i+1

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi-1)
        quick_sort(arr, pi+1, high)
    return arr

arr = list(map(int, input("Enter numbers separated by spaces: ").split()))
```

```
print("Sorted Array:", quick_sort(arr, 0, len(arr)-1))
```

6. Counting Sort

Description:
Counting sort is not a comparison sort and works by counting the number of objects having distinct key values. It's only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items.

Example:
Imagine students in a class have scored marks between 0 to 5. Instead of comparing marks, you just count how many have scored 0, how many scored 1, and so on. Then, you can easily construct the sorted list of marks.

Code:

```
def counting_sort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)
    output = [-1] * len(arr)
    for i in arr:
        count[i] += 1
    for i in range(1, len(count)):
        count[i] += count[i - 1]
    for i in range(len(arr) - 1, -1, -1):
        output[count[arr[i]] - 1] = arr[i]
        count[arr[i]] -= 1
    for i in range(len(arr)):
        arr[i] = output[i]
    return arr

arr = list(map(int, input("Enter numbers separated by spaces: ").split()))
print("Sorted Array:", counting_sort(arr))
```

These descriptions can be inserted alongside the codes in the lecture notes to give students both a conceptual understanding and a practical example of the algorithms.

---