CSCI 4470 Algorithms

Part I Foundations

- **1 The Role of Algorithms in Computing**
- **2 Getting Started**
- 3 Characterizing Running Times
- 4 Divide-and-Conquer
- 5 Probabilistic Analysis and Randomized Algorithms

Chapter 1 The Role of Algorithms in Computing

Chapter 2 Getting Started

```
1 The Role of Algorithms in Computing
  1.1 Algorithms
  1.2 Algorithms as a technology

2 Getting Started
  2.1 Insertion sort
  2.2 Analyzing algorithms
  2.3 Designing algorithms
```

6 Sorting Algorithms

1. **Bubble Sort**
2. **Selection Sort**
3. **Insertion Sort**
4. **Merge Sort**
5. **Quick Sort**
6. **Heap Sort**

Each of these sorting algorithms has its own set of characteristics, with varying performance in different scenarios.

Worst Case Analysis (usually):

- $T(n) =$ ***The max times on any input of size*** n
- Considering the T(n) a relation, not a function

1. Definition:

- Worst-case analysis evaluates an algorithm's maximum runtime for an input of size n.

2. Worst-Case Representation:

- $T(n)$ typically represents the running time of an algorithm for an input size of n.

3. Characteristics of $T(n)$:

- $T(n)$ ***is seen more as a relation rather than a fixed function***.
- Multiple function forms could describe an algorithm's time complexity, but in the context of the worst case, we are looking at the upper boundary or the worst performance.

4. Importance:

- By understanding the worst-case, one ensures the algorithm's performance will never exceed this boundary, providing assurance during its implementation.

5. Application Scenarios (New addition):

- For instance, when designing an application for real-time systems like air traffic control or medical equipment, understanding the worst-case runtime ensures that the system never breaches this time, ensuring safety and reliability.

Average Case Analysis (sometimes):

- $T(n) =$ ***The expected time over all input of size n***
- Requires an assumption of statistical distribution

1. Definition:

- Average case analysis estimates the expected running time of an algorithm across all possible input sets, giving an overall average performance perspective.

2. Mathematical Representation:

- For an input size of n, $T(n)$ represents the expected running time of an algorithm.

3. Assumptions and Constraints:

- To conduct an average case analysis, some assumptions about the statistical distribution of the inputs are required.
- Different input distributions might lead to different average-case results.

4. Importance:

- Average case analysis provides a better estimate of the real-world performance of an algorithm in regular use.
- It can assist decision-makers in choosing the right algorithm, especially in resource-constrained or performance-critical situations.

Best Case Analysis:

1. Definition:

- Best case analysis evaluates the running time of an algorithm under the most favorable input conditions. It provides the optimal performance an algorithm might achieve under stipulated conditions.

2. Representation:

- $T(n)$ denotes the algorithm's runtime under the most ideal input scenario for an input size of n.

2.1 Why Considered "Bogus" or "Cheat":

- Solely relying on best case analysis can be misleading. In real-world scenarios, it's rarely consistent with the best-case scenario.
- Relying on the best case to evaluate, promote, or sell an algorithm can be misleading to users. This is because, in real-world applications, it's rare for an algorithm to always operate under its best-case scenario.
- Choosing algorithms based solely on their best case might lead one to pick algorithms that aren't the best choice in most situations.

3. Importance:

- Even though best case analysis might not always be practical, it still can offer valuable insights into algorithm analysis, especially when one wants to know the lower bounds of algorithm performance.

4. Examination Points:

- Be able to define and identify the best case analysis.
- Understand why best case analysis is considered "bogus" or "cheat" in many contexts.
- Be capable of computing the best-case time complexity for specific algorithms.

5. Examples:

- For example, the Insertion Sort algorithm has a best-case time complexity of $O(n)$ when the list is already sorted.

Asymptotic Analysis and Asymptotic Notation

- Asymptotic analysis provides a way to understand the efficiency of algorithms without being tied to a specific hardware or environment. It focuses on the behavior of functions as their input sizes approach infinity.

What is the Insertion Sort's `WORSE-CASE` time?

- **Machine Dependency**: Algorithm performance can vary based on the type of computer and its relative or absolute speed. Asymptotic analysis abstracts these differences.
- Depends on Computers (types of computers)
- relative speed (on same machine)
- absolute speed (on different machines)

BIG IDEA! - Asymptotic Analysis

- Ignores machine-dependent constants.
- Focuses on the `Growth Rate` of $T(n)$ as n approaches infinity. $T(n)$ **as** $n \to \infty$

Asymptotic Notation

- **Theta** Θ - Notation, Represents both upper and lower bounds. **Theta** θ **(Notation)**
  - Drops non-dominant terms.
  - Ignore leading constants.

Example Asymptotic Notation:

Given $3n^3 + 90n^2 - 5n + 6046 = \theta(n^3)$, its asymptotic notation is $\theta(n^3)$ because as n grows, n^3 is the term that dominates the growth.

- As n approaches infinity, a $\theta(n^2)$ algorithm is faster than a $\theta(n^3)$ algorithm.

Practical Implication

- For large n, an algorithm with time complexity $\theta(n^2)$ will generally be faster than an algorithm with time complexity $\theta(n^3)$.

Insertion Sort Analysis

1. Definition and Characteristics

- **In-Place and Stable:** Insertion sort is an in-place and stable sorting algorithm.
- **Efficiency:** More efficient than bubble sort for small input sizes.
- **Pseudo code of Insertion Sort**:

```
INSERTION-SORT(A)
1 for i = 2 to A.length
2   key = A[i]
3       // Insert A[i] into the sorted subarray A[1 : i - 1].
4       j = i - 1
5   while j >= 1 and A[j] > key
6       A[j + 1] = A[j]
7       j = j - 1
8   A[j + 1] = key
```

Insertion Sort(A)	**cost**	**times**
$for j \leftarrow 2 \; to \; n$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
Insert $A[j]$ into the sorted sequence $A[1,...,j-1]$	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

2. Time Complexity Analysis

- **Worst-Case:** Occurs when the input is **reverse sorted**, with a time complexity of $O(n^2)$.
- **Best-Case:** Occurs when the input is **already sorted**, with a time complexity of $O(n)$.

2.1 Is Insertion Sort Fast?

- **Small n:** Moderately fast.
- **Large n:** Not suitable.

3. Reasoning Behind Time Complexity

- **Comparisons:** In the worst case, the total number of comparisons is given by the arithmetic series sum, which leads to $O(n^2)$ complexity.

$$0 + 1 + 2 + ... + (n-2) + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

- In W-C scenario, every time we try to insert a new element into the sorted part, we have to compare it against every element in the sorted part. So, the 1st element requires 0 comparisons, the 2nd requires 1, the 3rd requires 2, and so on, up to the last element which requires $n-1$ comparisons.

4. Importance of Understanding Worst-Case Complexity

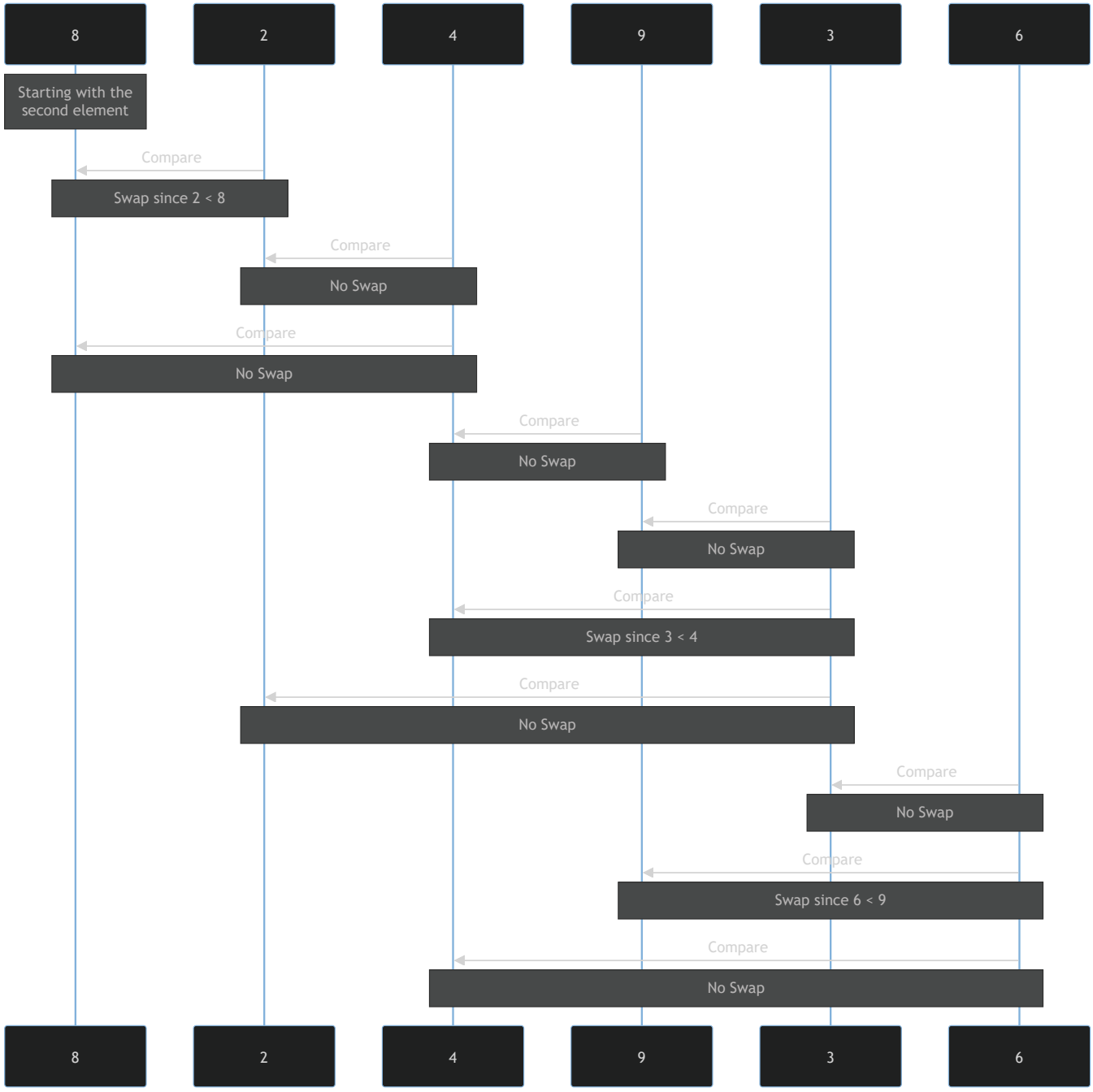
- **Performance Evaluation:** Crucial for evaluating its performance in specific scenarios, especially when the size of input data increases.

5. Examination Points:

- Be able to define Insertion Sort.
- Know the worst-case time complexity of Insertion Sort.
- Understand why the worst-case time complexity is $O(n^2)$.

6. Example a[8,2,4,9,3,6]:

Sure, the mermaid code to represent the demonstration of the insertion sort process for the array `a[8,2,4,9,3,6]` horizontally can be represented using a sequence diagram. Here's how you can illustrate it:



Loop Invariants & Correctness of Insertion Sort

Loop Invariants:

- **Definition**: A loop invariant is a property that holds before (and after) each iteration of a loop. Proving loop invariants can help establish the correctness of an algorithm.
- It is kind of a solution for algorithms

Three Essential Properties:

1. **Initialization**: The invariant holds true prior to the first loop iteration.

2. **Maintenance**: If the invariant holds true before any given iteration, it will continue to hold true before the next iteration.
3. **Termination**: Upon the loop's conclusion, the invariant gives a beneficial property that assists in demonstrating the algorithm's correctness.

Loop Invariant Example with Insertion Sort:

- Comparable to Mathematical Induction

Consider the array A = [8,2,4,9,3,6]. Let's see how the loop invariant proves the correctness of insertion sort.

Initialization:
Before the loop begins, the subarray contains only one element (in this case, A[1] or 8). A subarray with a single element is by default sorted.

Maintenance:
Assuming the elements from A[1] to A[i-1] are already sorted, when we consider the i-th element, the loop ensures it is placed in its correct position among the first i elements. This means, by the end of that iteration, elements from A[1] to A[i] will be sorted.

Termination:
After completing all n iterations, the loop invariant guarantees that the first n elements in the array are sorted. Hence, the whole array is sorted.

Key Points:

- Loop invariants are essential to prove algorithm correctness.
- The loop invariant concept can be applied to analyze various algorithms.

Examination Points:

- Definition of loop invariant.
- How to apply loop invariant in the Insertion Sort algorithm.
- For instance, when we move to A[2] (which is 2), it might get inserted before '8', making '2,8' the sorted subarray.

In-place Algorithm

Definition:
An in-place algorithm operates directly on its input and uses only a constant amount of extra memory.

Advantages:

- Minimal memory usage.

- Typically faster because they avoid the overhead of memory allocation and deallocation.

Drawbacks:

- Might modify or destroy the original data.
- Can be less intuitive and harder to implement in some scenarios.

Example using array a[2,5,10,3,5], Implementing a Insertion Sort:

Description:

Insertion Sort is an in-place, stable sorting algorithm that builds the final sorted array one item at a time. It is more efficient than Bubble Sort for smaller input sizes.

Example using array a[2,5,10,3,5]:

```
#include<iostream>
using namespace std;

void insertionSort(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int key = a[i];
        int j = i - 1;

        /* Move elements of a[0..i-1] that are greater than key
           to one position ahead of their current position */
        while (j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = key;
    }
}

int main() {
    int a[] = {2, 5, 10, 3, 5};
    int n = sizeof(a)/sizeof(a[0]);

    insertionSort(a, n);

    for(int i=0; i<n; i++)
        cout << a[i] << " ";

    return 0;
}
```

> Output will be: `2 3 5 5 10`

- Insertion Sort is also in-place as it only requires a constant amount of additional space. It directly modifies the input array `a`.

```
int key = a[i]; // key, the constant extra space
int j = i - 1; // j, the constant extra space

/* Move elements of a[0..i-1] that are greater than key
   to one position ahead of their current position */
while (j >= 0 && a[j] > key) {
    a[j + 1] = a[j];
    j = j - 1;
}
a[j + 1] = key;
```

1. **In-place Modification**: The code directly manipulates the array `a` without the need for another array to store the sorted result. It sorts the array by moving elements to their correct positions within the same array.
2. **Constant Extra Space**: The only additional variables we use during the sorting process are `key` and `j`. These both require a fixed (constant) amount of space regardless of the size `n` of the input. This means Insertion Sort runs using O(1) extra space.
3. In summary, these code snippets demonstrate how Insertion Sort operates directly on the original array, without the need for significant extra space, thereby achieving in-place sorting.

Implementing a Bubble Sort in-place:

```cpp
#include<iostream>
using namespace std;

void bubbleSort(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (a[j] > a[j+1]) {
                // Swap numbers in-place (原地交換兩個數字)
                int temp = a[j]; // temp needs the extra space
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}

int main() {
    int a[] = {2, 5, 10, 3, 5};
    int n = sizeof(a)/sizeof(a[0]);

    bubbleSort(a, n);

    for(int i=0; i<n; i++)
        cout << a[i] << " ";

    return 0;
}
```

Output will be: `2 3 5 5 10`

> The bubble sort here is in-place because it doesn't need any extra space larger than the input array `a` to sort. It operates directly on the input array.

Stable Algorithm

Definition:
A sorting algorithm is stable if identical elements retain their relative positions after sorting.

Importance:
Stability is crucial when sorting records with multiple fields. Without stability, records with the same key might mix up.

Example:
For records sorted by age with numbers as their IDs, if we sort an array like `a[5,7,10,5,4]`, a stable algorithm ensures the relative order of '5's remains unchanged.

Stable Sorting Algorithms:

- Bubble Sort
- Merge Sort
- Insertion Sort

Note: The stability might vary based on implementation.

Insertion Sort

Description:
Insertion Sort builds a sorted array one element at a time, akin to card sorting.

Steps:

1. Begin from the second element.
2. Compare with previous elements.
3. If it’s smaller, shift larger elements up until the correct position is found.
4. Insert the element.

Stability Demonstration:

With `a[5,7,10,5,4]`, Insertion Sort keeps the relative order of the '5’s unchanged.

Sorting Process

1. `[5,7,10,5,4]`
2. `[5,7,10,5,4]`
3. `[5,7,5,10,4]`
4. `[5,5,7,10,4]`
5. `[4,5,5,7,10]`