CSC 317: Data Structures and Algorithm Analysis

Dilip Sarkar

Department of Computer Science
University of Miami

Outline I

Outline of the Course Contents

- Algorithm Analysis
  - Complexity of Algorithms: space complexity and time complexity
  - Recurrence relations and methods for solving them
  - Expected complexity
- Algorithm Design Techniques
  - Divide and conquer
  - Dynamic programming
  - Greedy algorithms
- Advanced Data Structures
  - Hash Tables
  - Binary search trees
  - Red-black trees
  - Disjoint-sets
  - Representation of Graphs
- Graph Algorithms
  - Breadth-first and depth-first searches
  - Topological sorting and strongly connected components
  - Minimum spanning trees
  - Shortest paths

Five Questions

- What is an **Algorithm**?
- What is a **Data Structure**?
- What is a **Program**?
- What is a **Computational Problem**?
- What is a **Computational Complexity**?

Algorithms

- An **Algorithm** is a sequence of well-defined computational steps
- An algorithm may
  - takes some value or set of values as input
    - **Inputs**
  - produces some value or set of values as outputs
    - **Outputs**
- An Example: Sorting a sequence of integers
  - Input: $< 31, 41, 59, 26, 41, 58 >$
  - Output: $< 26, 31, 41, 41, 58, 59 >$
- **Correctness:** An algorithm is said to be **correct**, if **for every input** instance, it halts with a **correct output**
- An algorithm is for solving **a problem**.
- For solving a given **problem** there maybe **many** algorithms.
- Which one to use?
- Of course, a **fastest** algorithm.

Data Structures

- **Given:** a set of integers S_I and an integer i.
- **Problem:** is i in the set of integers S_I?
- Possible data structures for storing S_I,
  - Array in sorted order
  - Array unsorted
  - Linked-list
  - Binary search tree
  - AVL-tree
  - Red-black tree
  - B-tree, and
  - Hash table
- Which one is the best?
  - Sorted array, if binary search algorithm is used
- If elements are added to and/or deleted from S_I often, which data structure is the best?
- We answer above in *Algorithm Analysis* part.

Algorithm Analysis

- How to **determine computation time** of an algorithm?
  - Analyze the algorithm to determine memory and number of computational steps.
- There are two complexities to be considered
  - **Space complexity**: the **memory space** required to run the algorithm
    - Not the memory required for the inputs and outputs
  - **Time or Computational complexity**
  - Computation time depends on two factors:
    - speed of the computer, and
    - the algorithm
  - Computational complexity should NOT depend on computer's speed
  - Computational complexity should include **number of operations**
  - Three possible computational complexities
    - Best case – easy to compute, but too optimistic and not a good evaluation
    - Average case – usually very difficult to compute. we will do some
    - Worst case – yes, relatively easier to compute and unless otherwise said, **computational complexity** $\Longleftrightarrow$ **worst-case computational complexity**
- **Efficient algorithms**: Time complexity is a polynomial of input size n; $T(n) = O(n^k)$ for some fixed integer k.
- **Hard Problems:** They have no **known polynomial-time algorithm**.

Effect of Data Structure on Complexity: An example

Let us revisit the search problem with two additional operations:

- We have three operations on a **Dynamic Set** of Integers
- The operations are: `Find`, `Insert`, and `Delete`
- Let us consider only `Arrays` and `Linked lists`.

Data Structure	Operations		
	Find	Insert	Delete
Array (unsorted)	$O(n)$	$O(1)$	$O(n)$
Array (sorted)	$O(\log_2 n)$	$O(n)$	$O(n)$
Linked List (unsorted)	$O(n)$	$O(1)$	$O(n)$
Linked List (sorted)	$O(n)$	$O(n)$	$O(n)$

Table: Effect of data structure on time complexity: Find, Insert, Delete operations on a set of integers.

An Optimal Algorithm for a Problem

An algorithm is **optimal** if the time complexity of the algorithm is same as the *lower bound* of the problem.

- **Lower bound** on a **Problem**
  - A lower bound on a problem is minimum number of operation, in the worst-case, to solve the problem.
  - A lower bound can be developed 'algorithmically', or 'logically' without showing a computational algorithm
  - The **lower bound** is the worst (or maximum) of all the known lower bounds.
- An Example: Searching a set of n integers to find a given integer
  - $O(1)$ is lower bound, because at least one comparison is necessary to find the given integer
  - Binary search algorithm can find an integer (if it is present in the list) in $O(\log_2 n)$ comparisons if the element is in the list
  - Thus, the Lower bound for the searching problem is $O(\log_2 n)$ — the worst (or maximum) of $O(1)$ and $O(\log_2 n)$.
  - What is the minimum number of operations, if the integer is not presentation in the set?

An Optimal Algorithm for a Problem(cont.)

- **Optimal algorithm** for a **problem**: If the lower bound is same as the worst-case complexity of the algorithm, then the algorithm is an **optimal algorithm** for the **problem**.
- Two optimal algorithms for sorting are: Heap sort and merge sort
- **Quicksort** is not **an optimal algorithm** for the sorting problem
- Because complexity of standard **quicksort** algorithm $O(n^2)$
- **Upper bound on a Problem**
  - An *upper bound* on a problem is determined by known **fastest algorithms**
  - We need to consider worst-case complexities of all known algorithms.
  - The lowest worst-case complexity is the **upper bound.**
  - If upper bound is higher than the lower bound, then attempts are made to decrease the upper bound by designing better algorithms, or by reducing lower bound.

Algorithm as a technology

- Efficiency
- Insertion Sort Complexity: $c_{inst} n^2$
- Merge Sort Complexity: $c_{merg} n \log_2 n$
- Read Section 1.2

Insertion Sort: Illustration with cards

- The picture shows only one type of cards.
- Out of five cards, first three are sorted in increasing order
- After scanning to the right from third card
  - We find the smallest card is a '7' of clubs
  - Inset it at the 4th location
- We **don't have to worry about the last card**
- Nothing smaller than it to the right of it

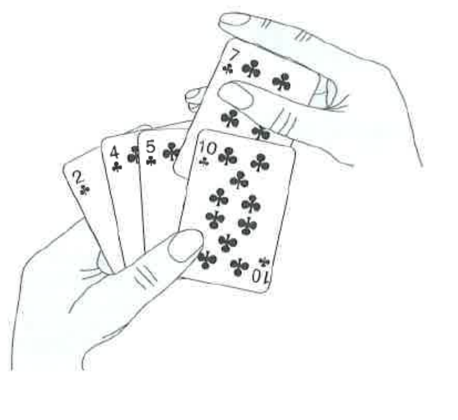
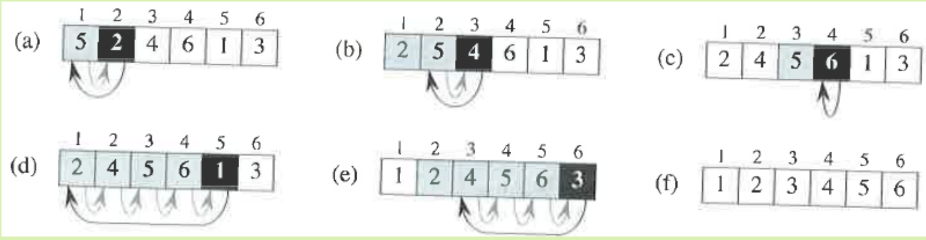


Figure 2.1 Sorting a hand of cards using insertion sort.

Insertion Sort: Example using an Array



Line 1: a loop from 2nd element to the last element

- Line 3: Assume A[1] to A[j-1] are sorted in non-decreasing order
- Line 2: Copy A[j] in *key*
- Line 4: Start from location i = (j-1)
- Lines 5 to 7: While *key* is greater than and NOT a the 1st element to the left
  - Line 6: Move A[i] to A[i+1]
  - Line 7: Next element is A[i-1]
- Line 8: Insert *key* to A[i+1]

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j - 1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

Figure: Pseudocode for Insertion Sort Algorithm

Insertion Sort: Loop Invariants and Correctness

- **Loop Invariant:**
  - Elements in A[1] to A[j-1] are same as the original array
  - But they are in sorted order
- We must show three things:
  1. **Initialization:** Loop invariant is true prior to the 1st iteration
  2. **Maintenance:** If it is true before an iteration of the loop, the invariant is true after the iteration
  3. **Termination:** When loop terminates, the invariant gives us a useful **property** that helps us to show that the algorithm is correct.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j - 1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

Figure: Pseudocode for Insertion Sort Algorithm

Proving Correctness using **loop invariant** is similar to **proof by induction**:

- Base case $\iff$ Initialization
- Induction Hypothesis $\iff$ Maintenance
- Induction Step $\iff$ Termination

Analyzing Algorithms: Example

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j - 1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

Figure: Pseudocode for Insertion Sort Algorithm

Line	Cost	times
Line 1	c_1	n
Line 2	c_2	$n-1$
Line 3	0	$n-1$
Line 4	c_4	$n-1$
Line 5	c_5	$\sum_{j=2}^{n} t_j$
Line 6	c_6	$\sum_{j=2}^{n}(t_j - 1)$
Line 7	c_7	$\sum_{j=2}^{n}(t_j - 1)$
Line 8	c_8	$n-1$

- Total cost $T(n)$ is calculated by summing costs of all 8 lines:
- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j) +$
  $c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$