CMPS 3120

Algorithm Analysis

Dr. Chengwei Lei
CEECS
California State University, Bakersfield
Example: sorting

- Input: A sequence of N numbers $a_1 \ldots a_n$
- Output: the permutation (reordering) of the input sequence such that $a_1 \leq a_2 \ldots \leq a_n$.
- Possible algorithms you’ve learned so far
  - Insertion, selection, bubble, quick, merge, ...
  - More in this course
- We seek algorithms that are both **correct** and **efficient**
Analysis of algorithms

- **Issues:**
  - correctness
  - time efficiency
  - space efficiency
  - optimality

- **Approaches:**
  - theoretical analysis
  - empirical analysis
Correctness

- What makes a sorting algorithm correct?
  - In the output sequence, the elements are ordered non-decreasingly
  - Each element in the input sequence has a unique appearance in the output sequence
    - [2 3 1] => [1 2 2]  X
    - [2 2 3 1] => [1 1 2 3]  X
Insertion Sort

InsertionSort(A, n) {
    for j = 2 to n {
        ▶ Pre condition: A[1..j-1] is sorted
        ▶ Post condition: A[1..j] is sorted
    }
}

1/31/2019
Insertion Sort

InsertionSort(A, n) {
    for j = 2 to n {
        key = A[j];
        i = j - 1;
        while (i > 0) and (A[i] > key) {
            A[i+1] = A[i];
            i = i - 1;
        }
        A[i+1] = key
    }
}
Example of insertion sort

1 2 4 5 6 1 3

2 5 4 6 1 3

2 4 5 6 1 3

2 4 5 6 1 3

1 2 4 5 6 3

1 2 3 4 5 6

Done!
Correctness

- What makes a sorting algorithm correct?
  - In the output sequence, the elements are ordered non-decreasingly
  - Each element in the input sequence has a unique appearance in the output sequence
    - \([2 3 1] \Rightarrow [1 2 2]\)  \(\times\)
    - \([2 2 3 1] \Rightarrow [1 1 2 3]\)  \(\times\)
Correctness

- For any algorithm, we must prove that it *always* returns the desired output for *all* legal instances of the problem.
- For sorting, this means even if (1) the input is *already sorted*, or (2) it contains *repeated elements*.
- Algorithm correctness is **NOT** obvious in some problems (e.g., optimization)
Use loop invariants to prove the correctness of loops

- A loop invariant (LI) is a formal statement about the variables in your program which holds true throughout the loop.

- **Claim:** at the start of each iteration of the for loop, the subarray \( A[1..j-1] \) consists of the elements originally in \( A[1..j-1] \) but in sorted order.

- **Proof** by induction
  - **Initialization:** the LI is true prior to the 1\(^{st}\) iteration.
  - **Maintenance:** if the LI is true before the \( j^{th}\) iteration, it remains true before the \( (j+1)^{th}\) iteration.
  - **Termination:** when the loop terminates, the LI gives us a useful property to show that the algorithm is correct.
Prove correctness using loop invariants

\[
\text{InsertionSort}(A, n) \{ \\
\text{for } j = 2 \text{ to } n \{ \\
\quad \text{key} = A[j] ; \\
\quad i = j - 1 ; \\
\quad \text{▷ Insert } A[j] \text{ into the sorted sequence } A[1..j-1] \\
\quad \text{while } (i > 0) \text{ and } (A[i] > key) \{ \\
\quad \quad A[i+1] = A[i] ; \\
\quad \quad i = i - 1 ; \\
\quad \} \\
\quad A[i+1] = \text{key} \\
\} \\
\}
\]

Loop invariant: at the start of each iteration of the for loop, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.
InsertionSort(A, n) {
    for j = 2 to n {
        key = A[j];
        i = j - 1;
        ▷ Insert A[j] into the sorted sequence A[1..j-1]
        while (i > 0) and (A[i] > key) {
            A[i+1] = A[i];
            i = i - 1;
        }
        A[i+1] = key
    }
}
InsertionSort(A, n) {
    for j = 2 to n {
        key = A[j];
        i = j - 1;
        ▷ Insert A[j] into the sorted sequence A[1..j-1]
        while (i > 0) and (A[i] > key) {
            A[i+1] = A[i];
            i = i - 1;
        }
        A[i+1] = key
    }
}
Termination

InsertionSort(A, n) {
    for j = 2 to n {
        key = A[j];
        i = j - 1;
        ▷ Insert A[j] into the sorted sequence A[1..j-1]
        while (i > 0) and (A[i] > key) {
            A[i+1] = A[i];
            i = i - 1;
        }
        A[i+1] = key
    }
}

The algorithm is correct!

The algorithm is correct!

Loop invariant: at the start of each iteration of the for loop, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.

Upon termination, A[1..n] contains all the original elements of A in sorted order.
Correctness alone is not sufficient
Brute-force algorithms exist for most problems
To sort $n$ numbers, we can enumerate all permutations of these numbers and test which permutation has the correct order

- Why cannot we do this?
- Too slow!
- By what standard?
How to measure complexity?

- Accurate running time is not a good measure
- It depends on input
- It depends on the machine you used and who implemented the algorithm
- It depends on the weather, maybe 😊
- We would like to have an analysis that does not depend on those factors