CS 2500: Algorithms

Lecture 10: Program Correctness and Sorting: Part II

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

September 19, 2024

Overview of Selection Sort

- Selection Sort is a comparison-based algorithm.
- It divides the input list into two parts:
  1. A sorted sublist of items which is built up from left to right.
  2. A sublist of the remaining unsorted items.
- The algorithm repeatedly selects the smallest element from the unsorted sublist, swaps it with the leftmost unsorted element, and moves the boundary between sorted and unsorted sublists by one element.

Selection Sort Algorithm

Algorithm:

1. Start with the first element (index 0) and assume it's the smallest.
2. Compare this element with the rest of the array to find the actual smallest element.
3. Swap the smallest element with the element at index 0.
4. Move to the next element and repeat the process for the remaining array.
5. Continue until the entire array is sorted.

Example: Selection Sort

Array: [29, 10, 14, 37, 14]

1. Find the minimum element in the array [29, 10, 14, 37, 14]. Minimum is 10.
2. Swap 10 with the first element (29): [10, 29, 14, 37, 14].
3. Find the minimum element in the remaining array [29, 14, 37, 14]. Minimum is 14.
4. Swap 14 with the second element (29): [10, 14, 29, 37, 14].
5. Repeat this process.

Selection Sort Algorithm

Algorithm SelectionSort(arr)

```
1:  n ← length of arr
2:  for i = 0 to n − 1 do                                  ▷ Outer Loop
3:      min_index ← i                          ▷ Assume i is the smallest
4:      for j = i + 1 to n − 1 do              ▷ Find the minimum element
5:          if arr[j] < arr[min_index] then
6:              min_index ← j
7:          end if
8:      end for
9:      Swap arr[i] and arr[min_index]         ▷ Place minimum at i
10: end for
```

Time Complexity

Best Case: $\mathcal{O}(n^2)$

- Even in the best case, Selection Sort performs $\mathcal{O}(n^2)$ comparisons.

Worst Case: $\mathcal{O}(n^2)$

- The algorithm always goes through $(n-1)$ comparisons for each element.

Space Complexity: $\mathcal{O}(1)$

- Selection Sort is an in-place sorting algorithm, requiring no additional memory for arrays.

Properties of Selection Sort

- **Stable:** No, because equal elements may be swapped in the process.
- **In-Place:** Yes, it uses constant extra memory.
- **Adaptive:** No, it always performs the same number of comparisons, regardless of the initial order of elements.
- **Suitability:** Good for small arrays where memory is a constraint, but inefficient for larger datasets.

Advantages and Disadvantages

Advantages:

- Simple to implement.
- Does not require extra memory, making it suitable for memory-constrained environments.

Disadvantages:

- Inefficient for large lists due to its time complexity of $\mathcal{O}(n^2)$.
- The algorithm does not adapt to the initial order of the elements.

Conclusion

- Selection Sort is a simple and intuitive algorithm.
- It is efficient for small data sets but not recommended for larger ones due to its time complexity.
- It provides insight into basic sorting mechanisms and helps understand more advanced sorting algorithms.

Insertion Sort

Intuition:

- Insertion Sort is similar to how you sort a hand of playing cards.
- You pick one card at a time and insert it into its correct position in an already sorted hand.
- The process is repeated for each new card until the entire hand is sorted.

Key Idea: As you add more cards, the left side of the hand is always sorted, and the right side contains the new unsorted cards.

Insertion Sort: Example

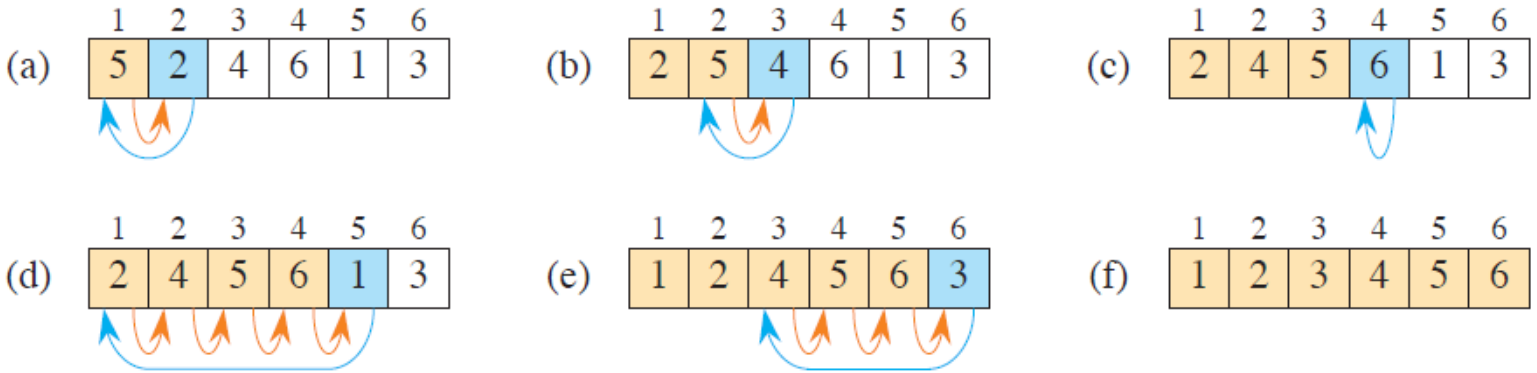

Figure: Visualization of Insertion Sort

Insertion Sort Algorithm

Algorithm:

Algorithm InsertionSort(arr)

```
1:  n ← length of arr
2:  for i = 2 to n do
3:      key ← arr[i]
4:      j ← i − 1
5:      while j > 0 and arr[j] > key do
6:          arr[j + 1] ← arr[j]    ▷ Shift larger elements right
7:          j ← j − 1
8:      end while
9:      arr[j + 1] ← key  ▷ Insert the key at correct position
10: end for
```

Time Complexity of Insertion Sort

- **Best Case:** $\mathcal{O}(n)$ when the array is already sorted.
- **Worst Case:** $\mathcal{O}(n^2)$ when the array is sorted in reverse order.
- **Average Case:** $\mathcal{O}(n^2)$.
- **Space Complexity:** $\mathcal{O}(1)$, as it only uses constant extra memory.

Properties of Insertion Sort

- **Stable:** Yes, equal elements are not swapped.
- **In-Place:** Yes, it uses constant extra memory.
- **Adaptive:** Yes, it becomes faster if the input is already partially sorted.

Advantages and Disadvantages of Insertion Sort

Advantages:

- Simple and intuitive to implement.
- Efficient for small or nearly sorted arrays.
- Minimal overhead and good for systems with memory constraints.

Disadvantages:

- Inefficient for large datasets due to its $\mathcal{O}(n^2)$ time complexity.
- Comparisons increase significantly as array size grows.

Conclusion

- Insertion Sort is a straightforward algorithm, best suited for small or partially sorted datasets.
- It works well in scenarios where the list is almost sorted or for small datasets.
- More efficient sorting algorithms like Merge Sort or Quick Sort are preferred for large datasets.

Loop Invariants and Algorithm Correctness

- A loop invariant is a condition that holds true before and after each iteration of a loop.
- To prove an algorithm correct using a loop invariant, we need to prove:
  - **Initialization:** The loop invariant is true before the first iteration.
  - **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
  - **Termination:** The loop terminates, and the invariant gives a useful property to show that the algorithm is correct.
- The proof is analogous to mathematical induction: prove a base case (Initialization) and an inductive step (Maintenance).

Proof of Correctness: Insertion Sort

Problem. Use the loop-invariant method to show that the algorithm for Insertion Sort is correct.

Proof of Correctness: Insertion Sort

Initialization

- We start by showing that the loop invariant holds before the first loop iteration when $i = 2$. The subarray $A[1 \ldots i-1]$ consists of just the first element $A[1]$.
- This subarray is trivially sorted since it contains only one element.
- Therefore, the loop invariant holds true before the first iteration.

Proof of Correctness: Insertion Sort

Maintenance

- Assume the loop invariant holds for $i = k$, meaning $A[1..k-1]$ is sorted.
- The inner `while` loop shifts elements greater than $A[k]$ to the right, making space to insert $A[k]$ in the correct position.
- After insertion, $A[1..k]$ is sorted, so the invariant is maintained.

Proof of Correctness: Insertion Sort

Termination

- The loop terminates when i exceeds n (i.e., when $i = n + 1$).
- At this point, the subarray $A[1 \ldots n]$ is sorted because the entire array has been processed.
- Since the loop invariant holds after the final iteration, the array is fully sorted when the loop terminates.
- This proves that the algorithm is correct.

Proof of Correctness: Insertion Sort

Conclusion

- Loop invariants provide a formal method to prove the correctness of algorithms.
- For Insertion Sort, we demonstrated correctness by proving:
  - The loop invariant holds at initialization.
  - It is maintained through each iteration.
  - At termination, the invariant ensures the entire array is sorted.
- This method of proof is useful for analyzing many types of iterative algorithms.

Proof of Correctness: Selection Sort

Initialization

- Before the first iteration ($i = 0$), the subarray $arr[0..n-1]$ is empty, which is trivially sorted.
- Therefore, the loop invariant holds before the first iteration.

Proof of Correctness: Selection Sort

Maintenance

- Assume the loop invariant holds at the start of iteration i.
- The inner loop finds the minimum element in the subarray $arr[i..n-1]$.
- This element is swapped with $arr[i]$, making $arr[0..i]$ sorted with the smallest $i+1$ elements.
- Thus, the invariant is maintained after each iteration.

Proof of Correctness: Selection Sort

Termination

- When the outer loop finishes, $i = n$, and the invariant implies that the subarray $arr[0..n-1]$ contains all elements in sorted order.
- Therefore, the array is fully sorted, proving the correctness of the Selection Sort algorithm.

Proof of Correctness: Selection Sort

Conclusion

- The loop invariant holds at initialization, is maintained throughout, and ensures correctness at termination.
- Therefore, the Selection Sort algorithm is correct.

Proof of Correctness: Bubble Sort

Initialization

- Before the first iteration ($i = 0$), no elements are considered to be sorted, which is trivially true since no swaps have been made yet.
- Therefore, the loop invariant holds before the first iteration.

Proof of Correctness: Bubble Sort

Maintenance

- Assume the loop invariant holds at the start of iteration i.
- The inner loop moves the largest unsorted element to its correct position at the end of the array.
- Thus, after each iteration of the outer loop, the last i elements are correctly sorted and are the largest elements.

Proof of Correctness: Bubble Sort

Termination

- When the outer loop finishes, $i = n$, meaning the entire array is sorted.
- Therefore, the array is fully sorted, proving the correctness of the Bubble Sort algorithm.

Proof of Correctness: Bubble Sort

Conclusion

- The loop invariant holds at initialization, is maintained throughout, and ensures correctness at termination.
- Therefore, the Bubble Sort algorithm is correct.

Proof of Correctness: Binary Search

Algorithm BinarySearch(arr, x)

```
1:  low ← 0
2:  high ← length of arr − 1
3:  while low ≤ high do
4:      mid ← (low + high) ÷ 2
5:      if arr[mid] = x then
6:          return mid                          ▷ Found the target
7:      else if arr[mid] < x then
8:          low ← mid + 1                       ▷ Search right half
9:      else
10:         high ← mid − 1                      ▷ Search left half
11:     end if
12: end while
13: return -1                                   ▷ Target not found
```

Proof of Correctness: Binary Search

Initialization

- Initially, $low = 0$ and $high = n - 1$, so the subarray $arr[low..high]$ is the entire array.
- If x is present, it must be in this subarray.
- Therefore, the loop invariant holds before the first iteration.

Proof of Correctness: Binary Search

Maintenance

- At each iteration, we compare $arr[mid]$ with x.
- If $arr[mid] < x$, we set $low = mid + 1$, reducing the search space to $arr[mid + 1..high]$.
- If $arr[mid] > x$, we set $high = mid - 1$, reducing the search space to $arr[low..mid - 1]$.
- In both cases, if x is present, it must still be within the updated subarray, maintaining the loop invariant.

Proof of Correctness: Binary Search

Termination

- The loop terminates when $low > high$. At this point, if x was found, it was returned; otherwise, the algorithm correctly concludes that x is not present.
- Therefore, the algorithm is correct.

Proof of Correctness: Binary Search

Conclusion

- The loop invariant holds at initialization, is maintained throughout, and ensures correctness at termination.
- Therefore, the Binary Search algorithm is correct.

Proof of Correctness: GCD

Algorithm GCD(a, b)

1: **while** $b \neq 0$ **do**
2: $\quad temp \leftarrow b$
3: $\quad b \leftarrow a \bmod b$ ▷ Remainder of a divided by b
4: $\quad a \leftarrow temp$
5: **end while**
6: **return** a ▷ a now contains the GCD of the original a and b

Proof of Correctness: GCD

Initialization

- Before the first iteration, the values of a and b are the original inputs.
- Since $gcd(a, b) = gcd(a, b)$, the loop invariant trivially holds.

Proof of Correctness: GCD

Maintenance

- At each iteration, we update b to $a\%b$ and a to the previous value of b.
- By the properties of the GCD, $gcd(a, b)$ remains the same after this transformation.
- Therefore, the loop invariant is maintained.

Proof of Correctness: GCD

Termination

- The loop terminates when $b = 0$. At this point, $gcd(a, 0) = a$, and by the loop invariant, a contains the greatest common divisor of the original input values.
- Therefore, the algorithm is correct.

Proof of Correctness: GCD

Conclusion

- The loop invariant holds at initialization, is maintained throughout, and ensures correctness at termination.
- Therefore, the Euclidean Algorithm for GCD is correct.