Foundations of Algorithm SCS1308

Dr. Dinuni Fernando PhD

Senior Lecturer

Merge Sort Algorithm

```
Merge-Sort(A,low,high)
if (low <high)
    mid = ceil(low+high)/2
    Merge-Sort(A, low, mid)
    Merge-Sort(A, mid+1, high)
    Merge(A, low, mid, high)
```

1. What is the runtime ?
2. Is it correct ?

```
Merge(A, low,mid,high)
L=A[low:mid]//(L is a new
array copied from A[low:mid])
R=A[mid+1,high]//(R is a new
array copied from
A[mid+1:high])
    i=1
    j=1
for k=low to high
If L[i] < R[j]:
    A[k] = L[i]
    i=i+1
else
    A[k] = R[j]
    j=j+1
```

Merge Sort Algorithm

- Uses divide and conquer programing paradigm.
- Divide Step
  - The array of size n is divided into two halves.
  - This step takes O(1) time as it involves simple index calculation.
- Conquer Step
  - The two halves are sorted recursively using the same algorithm.
  - Each recursive call process a subarray of size n/2
- Merge Step
  - The two sorted halves are merged together.
  - This step requires O(n) time as merging involves iterating through all elements of two subarrays.

Merge Sort Algorithm

- Let T(n) is the time complexity for sorting an array of size n
  - Divide Step – O(1)
  - Conquer Step – recursive calls for two subarrays of size n/2 contributing 2T(n/2)
  - Merge Step O(n)

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Merge Sort Algorithm

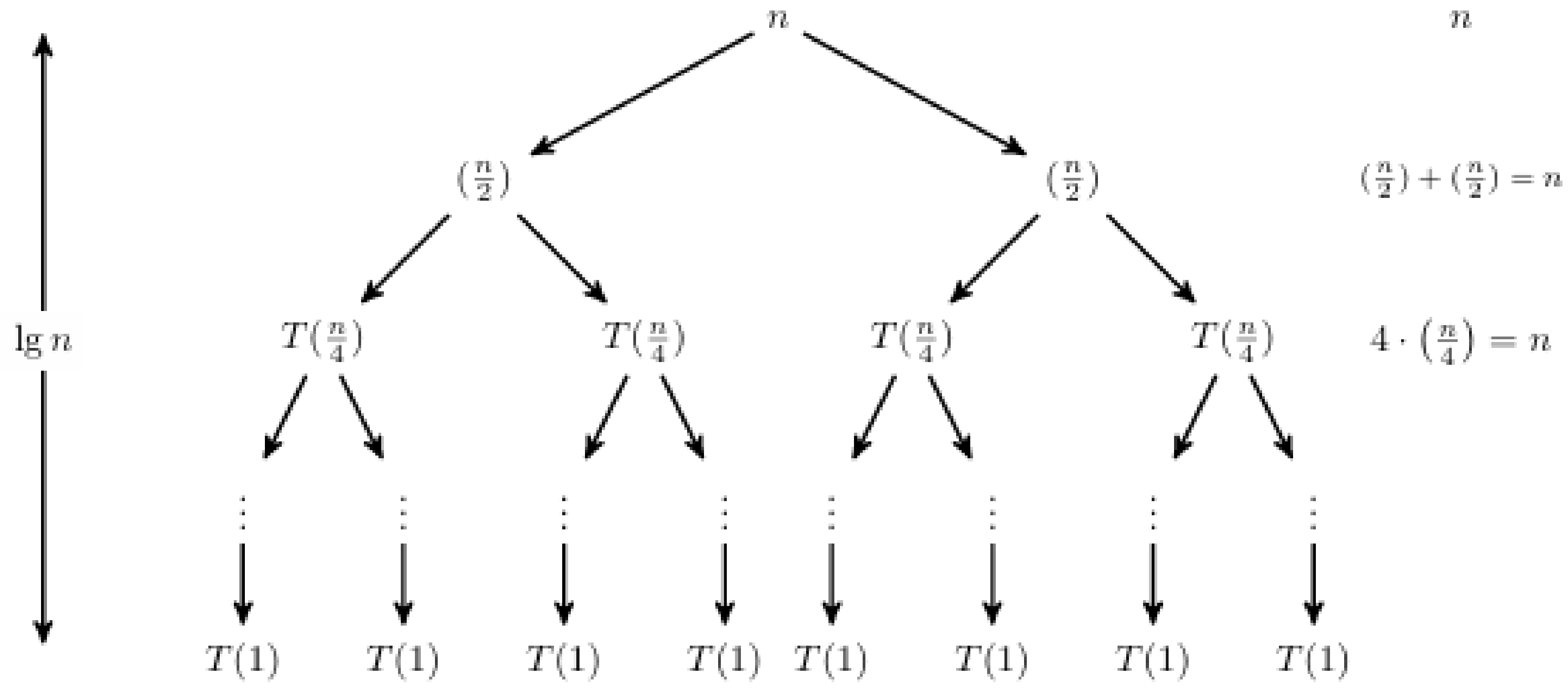
- Recurrence Relation

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases}$$

Draw the recursion tree !

Merge Sort Algorithm

$$T(n) = 2T(n/2) + n$$



Proving Correctness

- How to prove that an algorithm is correct ?
- Proof by:
  - Counterexample (indirect proof)
  - Induction (direct proof)
  - Loop Invariant
- Other approaches:
  - proof by cases/enumeration
  - proof by chain of iffs
  - proof by contradiction
  - proof by contrapositive

Proving 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 the input is already sorted, or it contains repeated elements.

Proof by Counterexample

Searching for counterexamples is the best way to disprove the correctness of some things.

- Identify a case for which something is NOT true.
- If the proof seems hard or tricky, sometimes a counterexample works.
- Sometimes a counterexample is just easy to see, and can shortcut a proof.
- If a counterexample is hard to find, a proof might be easier.

Proof by Induction

- Failure to find a counterexample to a given algorithm does not mean it is obvious that the algorithm is correct.
- Mathematical induction is a very useful method for proving the correctness of recursive algorithms.

1. Prove base case
2. Assume true for arbitrary value n
3. Prove true for case n +1

Proof by Induction – example

- Summing n integers : 1+2+3+4…….+n
- $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

Proof :

- Does it hold true for n =1 ?
- 1 = $\frac{1(1+1)}{2}$ ✓
- Assume it works for n ✓
- Prove that it's when n is replaced by n+1 ✓

Mathematical Induction

- Prove the formula for a base case
- Assume it's true for an arbitrary number of n
- Use the previous steps to prove that it's true for the next number n+1

Proof by Counterexample

- Definition : Proof by counterexample : Used to prove statements false, or algorithms either in correct or non-optimal.
- Prove or disprove: [x+y] = [x]+ [y] . – take the ceiling
  - Proof by counterexample: $x = \frac{1}{2}$ and $y = \frac{1}{2}$
- Prove or disprove: Every positive integer is the sum of two squares of integers
  - Proof by counterexample: 3
- Prove or disprove: $\forall x \forall y(xy \geq x)$ (over all integers)
  - Proof by counterexample: $x = -1 y = 3; xy = -3; -3 \ngeq -1$

Proof by Loop Invariant

- Built off proof by induction.
- Useful for algorithms that loop.
- Invariant : something that is always true

- Formally: find loop invariant, then prove:

1. Define a Loop Invariant : find candidate loop invariant, we prove
2. Initialization : How does the invariant get initialized ?
3. Maintenance : How does the invariant change at each pass through the loop ?
4. Termination : Does the loop stop ? When ?

Proving correctness

- Proof based on loop invariants
  - Loop invariant: An assertion which is satisfied before each iteration of a loop
  - At termination, the loop invariant provides important property that is used to show correctness
- Steps of proof:
  - Initialization (similar to induction base)
  - Maintenance (similar to induction proof)
  - Termination

More on the steps

- Initialization: Show loop invariant is true before (or at start of) the first execution of a loop.
- Maintenance: Show that if the loop invariant is true before an iteration of a loop, it is true before the next iteration
- Termination: When the loop terminates, the invariant gives us an important property that helps show the algorithm is correct.

So What's loop invariant ?

To analyze the correctness of the code using a **loop invariant**, we need to identify a property that holds true before and after every iteration of the loop.

Example 1: Finding maximum

```
Findmax(A, n)
    maximum = A[0];
    for (i = 1; i < n; i++)
     if (A[i] > maximum)
              maximum= A[i]
   return maximum
```

- What is a loop invariant for this code?

Proof of correctness

- Loop invariant for Findmax(A):

  "Before the i^{th} iteration (for $i = 1, \ldots, n$) of the for loop maximum = $max\{A[1],\ldots, A[i - 1]\}$"

Or

"At the start of each iteration of the loop (for index i), the variable maximum contains the largest value among the first i elements of the array A."

Initialization

- We need to show loop invariant is true at the start of the execution of the *for* loop

- Line 1: before the loop begins: sets maximum=A[0]
- At this point (i = 1), the subarray considered is just A [0]
- Since A[0] is the only element, it is indeed the maximum of the subarray.
- So the loop invariant is satisfied at the start of the *for* loop.

Maintenance

- Assume the loop invariant hosts at the start of the current iteration for some i;
- During the iteration, the algorithm compares A[i] with maximum
  - If A[i] > maximum, the value of maximum is updated to A[i]
  - Otherwise, maximum remains unchanged.

Maintenance (Cont'd)

- Assume that at the start of the i^{th} iteration of the *for* loop

  maximum = $max\{A[j] \mid j = 1, \ldots, i - 1\}$
- We will show that before the $(i + 1)^{th}$ iteration, maximum =max$\{A[j] \mid j = 1, \ldots, i\}$
- The code computes

  maximum=max(maximum, A[i]) = $max(max\{A[j] \mid j = 1, \ldots, i - 1\}, A[i])$ = max$\{A[j] \mid j = 1, \ldots, i\}$

Termination

- The loop terminates when i = n
- The loop invariant guarantees that the maximum contains the largest value among the first n elements of the array A.
- Since A has n elements, maximum is the largest values in entire A.
- So maximum = max{A[j]||j=1,...,n - 1}

Example 2: Linear Search

```
LinearSearch(A,v)
for j = 1 to A.length:
    if A[j] == v:
        return j
```

Example 2 : Loop Invariant

At the start of each iteration of the for loop (for index j), the algorithm has checked all elements in the subarray A[1...j-1].

If v is present in this subarray, it would have already been found, and the algorithm would have returned its index.

Otherwise, the search continues in the remaining array.

Example 2 : Initialization

- Before the loop begins (j = 1):
- The subarray considered is A[1...0], which is empty.
- Since there are no elements to check, the condition of the invariant holds trivially: there is no element v in the checked subarray, and the search is yet to begin.
- Thus, the loop invariant holds before the first iteration.

Example 2 : Maintenance

- Assume that the loop invariant holds at the start of the current iteration for some j.
- In the current iteration, the algorithm checks whether A[j] == v:
  - If A[j] == v, the algorithm returns j (index of the value v), which satisfies the correctness.
  - If A[j] != v, the algorithm continues to the next iteration.
- After the iteration, the invariant holds because:
  - The algorithm has now checked all elements in A[1...j].
  - If v is not found in this subarray, the search proceeds to A[j+1...].
- Thus, the loop invariant is maintained during each iteration.

Example 2 : Termination

The loop terminates when j = A.length + 1. At this point:

- All elements of the array A (i.e., A[1...A.length]) have been checked.
- If v was found in any iteration, the algorithm would have already returned its index.
- If v is not found, the loop terminates without returning, which signifies that v is not present in A.
- Thus, upon termination, the algorithm correctly concludes whether v exists in the array and, if so, returns its index.

Example 3 : Insertion sort

```
Insertion_Sort(A)
{
for (i = 1; i < n; i++)
    for (j = i; j >= 1 and a[j] < a[j-1]; j--)
          swap a[j] and a[j-1]
}
```

→ Loop invariant?

Example 3 : Loop invariant

- "At the start of each iteration of the outer loop (indexed by i), the subarray A[1...i-1] contains the same elements that were originally in A[1...i-1], but they are sorted in non-decreasing order."

- This invariant ensures that the portion of the array before index i is always sorted after each iteration.

Example 3 : Initialization

- Before the first iteration of the outer for loop (i = 1), the subarray A[1...i-1] is A[1...0]. This is an empty array.
- An empty array is trivially sorted.
- Thus, the loop invariant holds before the first iteration.

Example 3 : Maintenance

- At the start of an iteration of the outer loop (i):
  - By the loop invariant, the subarray A[1...i-1] is already sorted.
- During the inner while loop, the algorithm compares the current element A[j] with its predecessor (A[j-1]) and swaps them if they are out of order. This process "shifts" the element A[i] backward into its correct position.
- Once the inner loop completes:
  - The subarray A[1...i] becomes sorted while preserving all previously sorted elements.
- Thus, after each iteration of the outer loop, the loop invariant is maintained.

Example 3 : Termination

- The outer loop terminates when i = A.length + 1, meaning the entire array A[1...A.length] has been processed.
- By the loop invariant, the subarray A[1...i-1] is sorted. When i = A.length + 1, this subarray becomes the entire array A[1...A.length].
- Therefore, the entire array is sorted at the end of the algorithm.

Example 4 : Insertion Sort’

```
InsertionSort(A):
    for i = 1 to A.length:
        j = i
        while j > 0 and A[j-1] > A[j]:
            SWAP(A[j], A[j+1])
            j = j - 1
```

Example 4 : Insertion Sort’

```
InsertionSort(A):
    for i = 1 to A.length:
        j = i
        while j > 0 and A[j-1] > A[j]:
            # Mistakenly swap A[j] with A[j+1] instead of A[j-1]
            SWAP(A[j], A[j+1])
            j = j - 1
```

Example 4 : Expected Loop invariant

- "At the start of each iteration of the outer loop (indexed by i), the subarray A[1...i-1] is sorted in non-decreasing order."

- However, in this flawed code, the loop invariant is broken because the SWAP operation incorrectly swaps the current element A[j] with the next element A[j+1] instead of the previous element A[j-1].

Example 4 : Initialization

- Initialization: Before the first iteration, the invariant holds since A[1...0] is an empty array (trivially sorted).

Example 4 : Maintenance

- When the inner while loop is executed:

- The incorrect SWAP shifts the current element A[j] forward instead of backward, leaving the subarray A[1...i-1] unsorted.
- As a result, the elements in A[1...i-1] may no longer remain sorted after each iteration, violating the loop invariant.

Example 4 : Termination

- Since the loop invariant is violated during the execution, the final result is incorrect, and the array is not guaranteed to be sorted.

Example – Execution with Input

- Let the input array be A = [4, 2, 3, 1].
- Using the flawed code, the iterations proceed as follows:

- Iteration 1 (i = 1):
- Subarray A[1...1] is [4, 2].
- The while loop mistakenly swaps 4 with the next element (2) instead of the previous one, resulting in [4, 2, 3, 1].

- Iteration 2 (i = 2):
- Subarray A[1...2] is still unsorted. The same logic repeats.

Example 5 : Linear addition

```
sum =0;
for (i = 0; i < n; i++)
    sum = sum + A[i];
```

- What is a loop invariant for this code?

Example 5 : Loop invariant

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + A[i];
```

At the start of each iteration of the loop (before line 3), the value of sum is the sum of all elements in the array A from index 0 to i-1.

- Mathematically:

$$sum = A[0] + A[1] + ... + A[i-1]$$

Example 5 : Initialization

- Before the loop begins (when i = 0):

- The value of sum is initialized to 0.
- There are no elements summed yet, so the invariant holds true:
- sum = 0, which is the sum of the empty subset of A (from 0 to -1).

Example 5 : Maintenance

- During each iteration of the loop:

- The loop adds A[i] to sum, ensuring that after the iteration, sum reflects the sum of all elements from A[0] to A[i].
- Before the next iteration, i is incremented, so sum becomes the sum of elements from A[0] to A[i-1], maintaining the invariant.

Example 5 : Termination

- When the loop terminates (when i = n):

- The loop invariant ensures that sum is the sum of elements from A[0] to A[n-1].
- At this point, all elements of the array have been summed, and the program returns the correct total.

Example 6 : Bubble Sort

```
BubbleSort(A)
for i=1 to A.length-1
    for j=A.length to i+1
        if A[j]<A[j-1]
            Swap(A[j],A[j-1])
```

Example 6 : Bubble Sort

```
BubbleSort(A)
for i=1 to A.length 1                Outer loop
    for j=A.length to i+1            Inner loop
        if A[j]<A[j-1]
            Swap(A[j],A[j-1])
```

Example 6 : Expected Loop invariant [outer]

```
BubbleSort(A)
for i = 1 to A.length − 1
    for j = A.length to i + 1
        if A[j] < A[j − 1]
            SWAP(A[j], A[j − 1])
```

- At the start of the i^{th} iteration of the outer loop, the last i - 1 elements (A [1:i-1] are sorted and in their correct positions.

Example 6 : Initialization

- When i = 1, no elements have been processed yet, and the invariant holds trivially because no elements are in their correct sorted positions.
- Array A[1:i-1] is empty (i=1) and sorted by definition.

Example 6 : Maintenance

- Given the guarantees of the inner loop at the end of each iteration of the for loop at line 1, the value A[i] is the smallest values in the range A[i:A.range].
- Since the values in A[i:i-1] were sorted and were less than the value in A[i], the values in the range A[1:i] are sorted.

Example 6 : Termination

- The for loop at line 1 ends when i equals A.length-1. based on the maintenance proof, this means that all values in A[1:A.length-1] are sorted and less than the value at A[length]. So by definition A[1:A.length] are sorted.

- The invariant guarantees that every element is in its correct position.

Example 6 : Expected Loop invariant [inner]

```
BubbleSort(A)
for i = 1 to A.length − 1
    for j = A.length to i + 1
        if A[j] < A[j − 1]
            SWAP(A[j], A[j − 1]
```

At the start of each iteration of the inner loop (indexed by j), the largest element in the subarray A[j...A.length] is correctly positioned at the end of the subarray.

(A[A.length] after the first iteration, A[A.length - 1] after the second iteration, and so on).

Example 6 : Initialization

- Before the inner loop starts (j = A.length):
- The subarray A[j...A.length] contains only one element, A[A.length].
- A single-element array is trivially sorted, and the invariant holds.

Example 6 : Maintenance

- During each iteration, if A[j] < A[j-1], the two elements are swapped.
- This ensures that the smaller of the two elements moves closer to the start of the array, maintaining the invariant that elements from j to A.length are greater than or equal to A[j-1].
- As a result, the largest element in the subarray A[j...A.length] is guaranteed to move to the last position in that subarray.

Example 6 : Termination

- When the inner loop terminates (i.e., $j = i+1$), the largest unsorted element has been bubbled to its correct position, and the invariant guarantees that elements from j to A.length are sorted and greater than the preceding elements.

Example 7 : Merge sort

```
Merge-Sort(A,low,high)
if (low <high)
    mid = ceil(low+high)/2
    Merge-Sort(A, low, mid)
    Merge-Sort(A, mid+1, high)
    Merge(A, low, mid, high)
```

```
Merge(A, low,mid,high)
L=A[low:mid]//(L is a new array copied from A[low:mid])
R=A[mid+1,high]//(R is a new array copied from A[mid+1:high])
    i=1
    j=1
for k=low to high
If L[i] < R[j]:
    A[k] = L[i]
    i=i+1
else
    A[k] = R[j]
    j=j+1
```

Example 7 : Merge sort

```
Merge(A, low, mid, high)
L = A[low:mid] // (L is a new array copied from A[low:mid])
R = A[mid+1, high] // (R is a new array copied from A[mid+1, high])
i = 1, j = 1
for k =low  to high:
        if L[i] < R[j]:
                A[k] = L[i]
                i = i + 1
        else
                A[k] = R[j]
                j = j + 1
```

Example 7

- **Loop Invariant :** At the start of each for loop iteration, the array starting at A[k] with length k low contains the k low smallest elements, in increasing sorted order
- **Initialization** Prior to the rst iteration, the array starting at A[k] with length k low is empty because k low=0. L and R are assumed sorted.
- **Maintenance** Since L and R are sorted, the value at L[i] is the smallest in L and the value at R[j] is the smallest in R. The smallest of these is the smallest in the union of L and R, which is A[low : high]. Copy that into A[k].

Example 7

- **Termination** On the last iteration, k = high + 1. This means that the array at A[low] with length k low(low high+1) is sorted, which is the array A[low : high]. A[low high] is sorted.
- K-low = (high +1)-low = high low+1

Thank you