Erik Williams

epwilliams@csu.fullerton.edu

Project Report

The Lawnmower Algorithm

- is designed to solve the problem of organizing an array of alternating-colored disks, dark and light, into a sorted state where all dark disks precede all light disks. The algorithm's name reflects its operational pattern, simulating the back-and-forth motion of a lawnmower over a lawn.

- Initialization: The algorithm begins with an array (disks) of 2n elements, where n is the number of disks of each color (light and dark), arranged in an alternating pattern.

- Iterative Passes: The algorithm's core involves making (n+1)/2 passes(p) over the array. Each pass consists of two sweeps:

- Left to Right: The first sweep goes from the beginning of the array toward the end, comparing each pair of adjacent disks. They are swapped if a pair is found in the wrong order (a light disk preceding a dark disk). This action gradually moves light disks towards the right end of the array.

- Right to Left: Immediately following the left to right, a right to left is performed, starting from the second to last disk to the beginning. Like the first sweep, adjacent pairs are compared and swapped if necessary, pushing dark disks towards the left end.

- Repeated Application: These sweeps are repeated, narrowing the range of comparison with each pass to exclude the already correctly placed disks at the ends of the array. This optimization reduces the number of comparisons and swaps as the sort progresses, focusing the effort on the unsorted middle portion of the array.

- Termination: The algorithm terminates after completing the predetermined number of passes, by which point all the dark disks should be on the left side and all the light disks on the right side of the array, achieving a sorted state.

Mathematical Analysis

Outer Loop (for p from 0 to (n + 1) / 2):

This Loop runs approximately half the total disks, which means it operates for about n/2 iterations. This is because, after each complete pass (left to right and right to left), at least one disk of each color is in its final sorted position. Therefore, we don't need to iterate over these sorted elements in subsequent passes.

Inner Loop (for i from 0 to n - 2):

For each outer loop iteration, the inner loop runs from the start of the list to the second-to-last element, n-1 iterations. However, with each pass, the adequate size of the unsorted portion decreases. It might be made more efficient in the code by setting this Loop's upper bound to n - p - 1.

Swapping Condition (If disks[i] > disks[i + 1]):

A conditional check for whether a swap is needed within the inner Loop is performed. A swap is performed if the disk at the current index is greater than the disk at the following index (implies a light disk is before a dark disk). This is the essential operation of the sorting algorithm and is expected to occur less frequently as the list approaches a sorted state.

Reverse Inner Loop (for i from n - 2 down to 1):

Like the forward pass, but in reverse, starting from the second to last element and moving towards the second element. For optimization, the same reasoning that reduces the number of iterations after each outer loop iteration might be used here.

Time Complexity:

The algorithm has two nested loops. Each outer loop iteration triggers two inner loops of approximately n iterations each (forward and reverse). This gives us a time complexity of O(n^2). However, because the inner Loop's range is reduced with each outer loop iteration, the actual number of operations is less than n^2, but it remains quadratic.

Optimization Considerations:

The code might be optimized to break the inner loops earlier based on the sorting progress. This would not alter the overall O(n^2) complexity, but it would lower the constant factor in the time complexity.

The Alternate Algorithm

- is designed to sort a list of disks, initially arranged in an alternating pattern of two different types, into a configuration where similar types are grouped. This sorting process is achieved through a series of passes over the list, referred to as "r," where each r attempts to move the disks closer to the desired sorted state.

- **Initialization**: The algorithm starts with an array (disks) of $2n$ elements, where n represents the number of disks of each type. The disks are arranged in an alternating pattern.

- **Runs**: The algorithm consists of conducting n+1 runs over the array. An "i" is a single pass through the list, where specific pairs of adjacent disks are compared and potentially swapped to move them towards the correct position. The total number of r (n+1) ensures that the algorithm thoroughly processes each disk, accounting for the initial alternating arrangement.

- Starting Index: The algorithm calculates a starting index (s) using the modulo operation (i mod 2) for each run(i). This operation alternates the starting index between 0 and 1 for successive runs. The purpose of alternating the starting index(s) is to ensure that all pairs of disks are examined over two consecutive runs, as some pairs are skipped in a single pass due to the step of 2.

- Iteration and Swapping: The algorithm iterates over the array within each i, examining and potentially swapping pairs of adjacent disks. The iteration starts from the s and moves in steps of 2, checking every other disk. If a disk is found to be out of order relative to its neighbor, the algorithm swaps the two disks.

- Sorting: disks are compared directly, and swaps are made if the left disk should be on the right side according to the sorting goal.

- Outcome: After completing the specified runs(i), the algorithm results in the disks being sorted into two groups, each containing disks of a single type. Additionally, the function tracks and possibly returns the number of swaps performed, giving insight into the process's efficiency.

Mathematical Analysis

Initialization:

- The function inputs a list of disks and calculates its length n. This length defines the number of iterations for the sorting process.

Outer Loop (i loop):

- The outer Loop runs from 0 to n inclusive, resulting in n + 1 total iterations. This Loop controls the number of passes over the list.

Inner Loop (j loop):

- In each outer Loop iteration, the inner Loop starts from an index s, which is determined by the remainder when i is divided by 2. This means that on even iterations of the outer Loop, the inner loop starts at index 0, and on odd iterations, it starts at index 1.
- The inner Loop then steps through the list in increments of 2, effectively selecting every other element starting from the index s.
- This Loop runs until it reaches the second-to-last element of the list (n - 1), ensuring that the if condition can always compare disks[i] to disks[i + 1].

Swap Condition:

- Within the inner loop, there is a condition that checks the order of each pair of adjacent disks. If the disk at the current position, **disks[i]**, should be after the disk in the next position, **disks[i + 1]**, based on the sorting criteria, then the two disks are swapped.
- The swap operation is crucial for the algorithm because it incrementally moves the disks into the correct order with each pass. Specifically, it ensures that disks that are out of order are progressively moved towards their final sorted position with each iteration of the loop.

Time Complexity:

- Within the inner loop, there is a condition that checks the order of each pair of adjacent disks. If the disk at the current position, **disks[i]**, should be after the disk in the next position, **disks[i + 1]**, based on the sorting criteria, then the two disks are swapped..
- The swap operation is crucial for the algorithm because it incrementally moves the disks into the correct order with each pass. Specifically, it ensures that disks that are out of order are progressively moved towards their final sorted position with each iteration of the loop.

Efficiency:

- The AlternateSort function follows a quadratic time complexity (O(n^2)), meaning its performance degrades quickly as the list size increases.

In summary, An array of alternating colored disks is efficiently sorted using the Lawnmower Algorithm into a sequence where one color comes before the other. It arranges the disks by going back and forth across the array like a lawnmower. As the algorithm advances, key optimizations include minimizing swaps and comparisons by concentrating on unsorted sections. Next, the Alternate Algorithm modifies its starting index to guarantee comparison to cluster comparable disks together via passes. Both algorithms exhibit quadratic time complexity; however, there are opportunities to improve efficiency by utilizing computing techniques to boost speed or decrease the range of comparison as sorting progresses.

```
Erik_Williams ~/project-1-implementing-algorithms-erik-williams$ make
g++ -std=c++11 -Wall disks_test.cpp -o disks_test
./disks_test
disk_state still works: passed, score 1/1
sorted_disks still works: passed, score 1/1
disk_state::is_initialized: passed, score 1/1
disk_state::is_sorted: passed, score 1/1
alternate, n=3: passed, score 1/1
alternate, n=4: passed, score 1/1
alternate, other values: passed, score 1/1
lawnmower, n=3: passed, score 1/1
lawnmower, n=4: passed, score 1/1
lawnmower, other values: passed, score 1/1
TOTAL SCORE = 10 / 10
```

```
Function Lawnmower(disks: List)
    n = length(disks)

    for p from 0 to (n + 1) / 2  do
        for i from 0 to n - 2 do
            if disks[i] > disks[i + 1] then
                swap(disks[i], disks[i + 1])
        end for

        for j from n - 2 down to 1 do
            if disks[j] < disks[j - 1] then
                swap(disks[j - 1], disks[j])
        end for
    end for
End Function


Function AlternateSort(disks: List)
    n = length(disks)

    for i from 0 to n
        s = i mod 2

        for j from s to n - 1 step 2 do
            if disks[j] > disks[j + 1] then
                swap(disks[j], disks[j + 1])
        end for
    end for
End Function
```

Lawnmower Algorithm

```
Function Lawnmower (disks: List)
  n = length(disks)

  for p from 0 to (n+1)//2 :
    for i from 0 to n-2 :
      if disks[i] > disks[i+1] :
        swap(disks[i], disks[i+1])

    for j from n-2 down to 1 :
      if disks[j] < disks[j-1] :
        swap(disks[j-1], disks[j])
End Function
```

$$\sum_{p=0}^{\frac{n+1}{2}+1} \sum_{i=0}^{n-2} 2 \quad \sum_{j=0}^{n-2+1} 2$$

$n-2+1 \times 2 = n$

$\frac{n+1}{2}+1 = \frac{1}{2}n + \frac{3}{2}$

$n-2+1-1 \times 2 = n-3$

$n \times \frac{1}{2}n + \frac{3}{2} \times n - 3$

$\frac{1}{2}n^2 + \frac{3}{2}n - 3$

$= \frac{1}{2}n^2 + \frac{3}{2}n - 3 \rightarrow O(n^2)$

Alternate Algorithm

```
Function Alternate Sort(disks: List)
    n = length(disks) - 1
    for i from 0 to n do - n+1
     s = i % 2 - 2

      for j from 0 to n-1 step 2 do
        if disks[j] > disks[j+1] then - 1
  1 step → swap_(disks[j], disks[j+1]) - 1
      end for
    end for
  End Function
```

step size 2

$$\sum_{j=0}^{n-1} 2 \rightarrow \frac{n-1}{2}$$

$$\sum_{i=0}^{n+1} 2 = 2(n+1) = 2n+2$$

$$\frac{n-1}{2} * 2 \times 2n+2$$

$$\left(\frac{1}{2}n - \frac{1}{2}\right) \cdot 2 \cdot 2n + 2$$

$$((2 \cdot 2n) \cdot \tfrac{1}{2}n - (2 \cdot 2n) \cdot \tfrac{1}{2}) + 2$$

$$(2n^2 - 2n) + 2$$

$$2n^2 - 2n - 2 \rightarrow O(n^2)$$