ENPM809X: Data and Algorithms: Homework #1

Due on February 09, 2023 at 4:00pm

Section 0101

Mahima Arora

Problem 1

2.2(b-d) Bubble Sort and related three parts(refer to HW1 for full questions).

Solution

PART B
Loop invariant for loop in lines 2-4 is as follows:
At the start of each iteration, the sub-array A[j....n] consists of elements originally in that sub-array before entering the loop but in a different order and the first element is the smallest among them.

1. **Verify initialization**: Initially, array A[1] is the single original element and it is sorted.
2. **Verify iteration**: In each iteration, A[j] is compared with A[j-1] making A[j-1] the smallest in the sub-array. After the jth iteration, the largest j elements are moved to the end of the array making A.size() - i - 1 element sorted.
3. **Verify termination**: This loop terminates when j = i + 1 and the sorted array is A[1...n] which is the entire input.

PART C
Loop invariant for loop in lines 1-4 is as follows:
At the start of each iteration for the first for loop, the sub-array A[1....i-1] consists of elements that are smaller than elements in sub-array A[i...n] in sorted order(first i-1 numbers will be sorted after i iterations).

1. **Verify initialization**: Initially, array A[1...i-1] is empty and hence, the smallest element of the sub-array.
2. **Verify iteration**: After execution of the inner loop, A[i] will be smallest of the sub-array A[i....n] and in the start of the outer loop, A[1....i-1] sub-array consists of elements smaller than elements of A[i....n] in sorted order.
3. **Verify termination**: This loop terminates when i = A.length - 1 and the sorted array is A[1...n] which is the entire input.

PART D The Worst-case running time of Bubble Sort is $\Theta(n^2)$ if the array is reversed. Insertion Sort has a running time that varies from $\Theta(n)$ to $\Theta(n^2)$. Comparing both, Insertion Sort is slightly better than bubble sort as insertion sort has fewer swaps as compared to bubble sort.

Problem 2

2.3-4(a-c) Recursive process for Insertion Sort including pseudocode, running time, and solving recurrence to find the complexity(refer to HW1 for full questions).

Solution

Part A

Pseudocode implementation of the recursive algorithm:

Insertion-Sort-Recursion($Array, n$**)**

Algorithm 1 Recursive Insertion Sort

```
function INSERTION-SORT-RECURSION(Array, n)
    if n ≥ 1 then
        INSERTION-SORT-RECURSION(Array, n − 1)
        SUBSTITUTE(Array, n)
    end if
end function
function SUBSTITUTE(Array, m)
    key ← Array[m]
    idx ← m − 1
    while idx > 1ANDArray[idx] > key do
        Array[idx + 1] ← Array[idx]
        idx ← idx − 1
    end while
    Array[idx + 1] ← Array[idx]
end function
```

Part B

Recurrence for the running time of the recursive insertion sort:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1. \\ T(n-1) + \Theta(n), & \text{if } n > 1. \end{cases} \tag{1}$$

Part C

Solving the recurrence, time complexity comes out to be: $\Theta(n^2)$. Let $\Theta(n) = n$. Substituting this value in the equation above, we get

$$\begin{aligned} T(n) &= T(n-1) + n \\ T(n-1) &= T(n-2) + n - 1 \\ T(n) &= T(n-2) + 2n - 1 \\ &\vdots \\ T(n) &= T(n-k) + kn - \frac{(k-1)k}{2} \end{aligned}$$

Problem 2 continued on next page. . .

Setting k = n - 1,

$$\begin{aligned}T(n) = T(n-n+1) + (n-1)n - \frac{(n-1-1)(n-1)}{2} \\ T(n) = T(1) + (n-1)n - \frac{(n-2)(n-1)}{2} \\ T(n) = T(1) + \frac{(n+2)(n-1)}{2} \\ T(n) = \mathcal{O}(n^2)\end{aligned}$$

Problem 3

4.1-5 Use the following ideas to develop a non-recursive, linear-time algorithm for the maximum sub-array problem: Start at the left end of the array, and progress toward the right, keeping track of the maximum sub-array seen so far. Knowing a maximum sub-array of A[1....j], extend the answer to find a maximum sub-array ending at index j+1 by using the following observation: A maximum sub-array of A[1.. j+1] is either a maximum sub-array of A[1.. j] or a sub-array A[i.. j+1], for some $1 \leq i \leq j+1$. Determine a maximum sub-array of the form A[i.. j+1] in constant time based on knowing a maximum sub-array ending at index j.

Solution

We can use Kadane's Algorithm to get a linear running time. The algorithm is as follows:

find-max-subarray(*A*)

Algorithm 2 Kadane's Algorithm for Linear Time

function FIND-MAX-SUBARRAY(A)
  $maxSum \leftarrow nums[0]$
  $i \leftarrow 1$
  while $i < A.length$ **do**
    $nums[i] \leftarrow max(nums[i], nums[i] + nums[i-1])$
    $maxSum \leftarrow max(nums[i], maxSum)$
  end while
return $maxSum$
end function

This algorithm returns the max sub-array for a given array in Linear time. The max sum is stored in the variable called maxSum and after each iteration, it updates the [i-1] value with the latest sum. It keeps comparing this [i-1] value with the maxSum and stores the greater value in the same variable, hence, always keeping track of max-subarray in A[1.....j].

Problem 4

4.5-1 Use the master method to give tight asymptotic bounds for the following recurrences. **Solution**

Part A

$T(n) = 2T(n/4) + 1$
The answer is $\Theta(\sqrt{n})$**.**

Since $f(n) = 1$ and $T(n) = n^{\log_b a}$ where a = 2 and b = 4. Therefore,

$$f(n) = n^{\log_4 2 - 1/2} = 1$$
$$T(n) = n^{\log_4 2} = \sqrt{n}$$

Thus Case I of the Master Method applies where T(n) is more dominant than f(n).

Part B

$T(n) = 2T(n/4) + \sqrt{n}$
The answer is $\Theta(\sqrt{n}\log n)$**.**

Since $f(n) = \sqrt{n}$ and $T(n) = n^{\log_b a}$ where a = 2 and b = 4. Therefore,

$$f(n) = n^{\log_4 2} = \sqrt{n}$$
$$T(n) = n^{\log_4 2} = \sqrt{n}$$

Thus Case II of the Master Method applies where T(n) equals f(n).

Part C

$T(n) = 2T(n/4) + n$
The answer is $\Theta(n)$**.**

Since $f(n) = n$ and $T(n) = n^{\log_b a}$ where a = 2 and b = 4. Therefore,

$$f(n) = n^{\log_4 2 + 1/2} = n$$
$$T(n) = n^{\log_4 2} = \sqrt{n}$$

Thus Case III of the Master Method applies where f(n) is more dominant than T(n).

Part D

$T(n) = 2T(n/4) + n^2$
The answer is $\Theta(n^2)$**.**

Since $f(n) = n^2$ and $T(n) = n^{\log_b a}$ where a = 2 and b = 4. Therefore,

$$f(n) = n^{\log_4 2 + 3/2} = n^2$$
$$T(n) = n^{\log_4 2} = \sqrt{n}$$

Thus Case III of the Master Method applies where f(n) is more dominant than T(n).