Name: Inchara Raveendra

SCU ID: 00001653600

Assignment -1

2.2-1

Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

Highest order term of the function is n^3.
Constant co-efficients are ignored because they are less significant in terms of rate of growth while calculating computational efficien for inputs of larger value
Therefore for $f(n) = \frac{n^3}{1000} - 100n^2 - 100n + 3$, $\Theta(n^3)$

2.2-2

Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n - 1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n - 1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

Pseudocode

```
SELECTION-SORT(A)
1    for i = 1 to A.length - 1
2        minIndex = i
3        for j = i + 1 to A.length
```

```
4          if A[j] < A[minIndex] and j ≠ minIndex
5              minIndex = j
6      swap A[i] with A[minIndex]
```

Loop invariant

When each iteration of the outer 'for' loop begins, the subarray $A[1 \ldots i-1]$ contains $i-1$ smallest elements of A which are sorted in increasing order

Why does it run for only the first n-1 elements rather than all 'n' elements?

This is because the minimum element in $A[1 \ldots n-1]$ will be compared with $A[n]$ in the last iteration and will be swapped, if needed. So, there is no need to continue the algorithm till the last element.

Running time

Line	Cost	Time
1	c_1	n

2	C_2	$n-1$
3	C_3	$\sum_{j=2}^{n} t_j$
4	C_4	$\sum_{j=2}^{n} (t_j - 1)$
5	C_5	$\sum_{j=2}^{n} (t_j - 1)$
6	C_6	$n-1$

Considering a reverse-sorted array for worst-case. Inner 'for' loop is executed t_j times for $j = 2, 3, \ldots, n$ where $n = A.length$

Line 5 will be executed less than $t_j - 1$ times in the average case but it is still the order of n. For worst case, it'll be executed exactly $t_j - 1$ times.

$$t_j = (n - j) + 1$$

$$\sum_{j=2}^{n} t_j = (n-2)+1 + (n-3)+1 + \ldots\ldots + 1$$

$$= (n-1) + (n-2) + \ldots\ldots + 1$$

$$= \frac{(n-1)(n)}{2}$$ from arithmetic progression summation

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} - \sum_{j=2}^{n} 1$$

$$= \frac{n(n-1)}{2} - \left[\sum_{j=1}^{n} 1 - 1\right] = \frac{n(n-1)}{2} - (n-1)$$

$$= \frac{n(n-1) - 2(n-1)}{2}$$

$$= \frac{(n-1)(n-2)}{2}$$

Running time can be calculated as follows:

$$T(n) = c_1(n-1) + (c_2 + c_6)n + c_3 \frac{n(n-1)}{2}$$

$$+ (c_4 + c_5) \frac{(n-1)(n-2)}{2} \rightarrow (1)$$

∴ Worst case running time is $\Theta(n^2)$

For best-case scenario, array is already sorted in increasing order. Therefore, line 5 will not be executed $\Rightarrow c_5 = 0$

Although $c_5 = 0$ in equation (1),

$$T(n) = c_1(n-1) + (c_2 + c_6)n + c_3 \frac{n(n-1)}{2}$$

$$+ c_4 \frac{(n-1)(n-2)}{2}$$

∴ Best case running time is also $\Theta(n^2)$

2.3-3

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

Base Case

When $n = 2$, $T(2) = 2 = 2 \log_2 2$. So, the solution is true

Induction

Suppose it is true for k, we will prove that it is true for k+1

$$T(n) = 2T(n/2) + n \quad \text{if } n = 2^k, k > 1$$

$$\therefore \quad T(2^k) = 2T\left(\frac{2^k}{2}\right) + 2^k = 2^k \log_2 2^k \rightarrow (1)$$

For $k = k+1$,

$$T(2^{k+1}) = 2T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1}$$

$$= 2T(2^k) + 2^{k+1}$$

$$= 2\left[T(2^k) + 2^k\right]$$

$$= 2\left[2^k \log_2 2^k + 2^k\right]$$

$$= 2^{k+1}\left[\log_2 2^k + 1\right]$$

$$= 2^{k+1}\left[\log_2 2^k + \log_2 2\right]$$

$$= 2^{k+1} \log_2 2^{k+1}$$

$\therefore$ $T(n) = n \log_2 n$ holds for all n that are exact powers of 2

2-2 Correctness of bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

a. Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \le A'[2] \le \cdots \le A'[n] , \tag{2.3}$$

where $n = A.length$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.3).

a) We have to prove that A' consists of same elements as A, which is true since A' is obtained by swapping the elements of A. So, the output array A' must contain all the elements present in array A.

b. State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

b) Loop invariant for inner loop

For the 'for' loop in lines 2-4,

At the start of each iteration of the loop, the subarray $A[j \ldots n]$ contains elements in the main array $A[j \ldots n]$ but in a different order and the first element is the smallest among them.

Initialization : Initially, the subarray consists of only $A[n]$ and this is the smallest element of the subarray.

Maintenance : In every step, we compare $A[j]$ and $A[j-1]$ and $A[j-1]$ becomes the smallest term among them. After every iteration, the length of the sub-array increases by one and first element is always the smallest term.

Termination : The loop ends when $j = i+1$ and length of sub-array increases by one and the first element is the smallest term in the subarray as we swap $A[i+1]$ with $A[i]$

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

c) Loop invariant for outer for loop

For the 'for' loop in lines 1-4,
At the start of each iteration, the subarray $A[1 \ldots i-1]$ contains elements which are smaller than the elements in the subarray $A[i \ldots n]$ and it is sorted.

Initialization : Initially, the subarray $A[1 \ldots i-1]$ has zero elements

Maintenance : Once inner for loop has been executed, $A[i]$ will be the smallest term of $A[i \ldots n]$ and when outer loop starts, $A[1 \ldots i-1]$ contains elements smaller than

elements in $A[i.....n]$ in a sorted order. After outer loop has been executed, $A[1....i]$ contains elements smaller than $A[i+1.....n]$ in a sorted order

Termination : The loop ends when $i = A.length$ and $A[1....n]$ contains all elements in a sorted order

d. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

Worst case scenario → Reverse sorted array
Bubble sort iterates over the entire array and performs 'n' comparisons and swaps. So,

# of terms x # of comparisons = n^2

and worst-case running time of bubble sort is $\Theta(n^2)$

Worst-case running time for Insertion Sort is also $\Theta(n^2)$. However, number of swaps are less and that implies lesser constant factors. So, insertion sort is faster than Bubble sort for a given input size.

2-4 *Inversions*

Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

a) $(1, 5)$

$(2, 5)$

$(3, 4)$

$(3, 5)$

$(4, 5)$

b. What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

b) $\{n, n-1, \ldots\ldots, 2, 1\}$ has the most inversions since its in a reverse-sorted order

Total number of inversions = Number of ways to choose 2 distinct integers

$= {}^{n}C_2$

$= \frac{n(n-1)}{2}$

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

c) Maximum number of inversions are possible when the array is reverse-sorted. Therefore, the higher the number of inversions in an array, longer is the time taken by insertion sort to sort the array

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

d) Running time with log n would mean divide-and-conquer algorithms
Similar to merge sort, the array has to be divided into halves recursively and number of inversions can be counted.
This happens in log n steps and $\theta(n)$ operations to count the inversions. Combining the two, $\theta(n \log n)$

Pseudocode

```
MODIFIED-MERGE-SORT (A, p, r)
    if (p < r) then
        q = ⌊(p+r)/2⌋
        left  = MODIFIED-MERGE-SORT (A, p, q)
        right = MODIFIED-MERGE-SORT (A, q+1, r)
        inv   = MODIFIED-MERGE-SORT (A, p, q, r) + left + right
        return inv
```

```
MODIFIED-MERGE-SORT (A, p, q, r)
    inv = 0
    n1 = q - p + 1
    n2 = r - q
    let L[1, .... n1] and R[1 .... n2] be new arrays
    for i = 1 to n1
        L[i] = A[p + i - 1]
    for j = 1 to n2
        R[j] = A[q + j]
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1
    j = 1
    for k = p to r
```

```
    if L[i] ≤ R[j] then
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        inv = inv + (n1 - i + 1)
        j = j + 1
return inv
```