Sardar Vallabhbhai National Institute of Technology, Department of Computer Science and Engineering, Surat

September 12, 2023

1.

(a) $f(n) = 100 * 2^n + 8n^2$

To prove: $f(n) = O(2^n)$

$f(n) \leq 100 * 2^n + 8 * 2^n$
$f(n) \leq 108 * 2^n$ ...(A)
Here, $c_1 = 108$ and $g(n) = 2^n$ and $n_0 = 1$
$\therefore \boldsymbol{f(n) = O(2^n)}$

Also, to check if $f(n) = \theta(2^n)$

$f(n) \geq 100 * 2^n + 8 * (0)$
$f(n) \geq 100 * 2^n$ ...(B)
Here, $c_2 = 100$ and $g(n) = 2^n$ and $n_0 = 1$
$\therefore \boldsymbol{f(n) = \Omega(2^n)}$

From (A) and (B):
$0 \leq 108 * 2^n \leq 100 * 2^n + 8n^2 \leq 100 * 2^n$ for all $n \geq n_0$
i.e. $c_1 g(n) \leq f(n) \leq c_2 g(n)$ where $g(n) = 2^n$
$\therefore \boldsymbol{f(n) = \theta(2^n)}$

(b) $f(n) = 3n + 8$
$f(n) \geq 3n$
Here, $c = 3$ and $n_0 = 1$

$\therefore \boldsymbol{f(n) = \Omega(n)}$

Given:$f(n) = 3n + 3 = \Omega(n)$
$f(n) = 3n + 3 = \Omega(n)$
Both the above facts are correct. Ω defines the lower bound of an algorithm. However, Ω is not tightly-bound. Thus, all the set of functions with their growth rate lower than that of the actual lower bound are included in the lower bound of the function.

$f(n) = 3n + 3$
$f(n) \geq 3n$
Here, $c = 3$ and $n_0 = 1$
$\therefore f(n) = \Omega(n)$

$f(n) = 3n + 3$
$f(n) \geq 3 * 1$
Here, $c = 3$ and $g(n) = 1$
$\therefore f(n) = \Omega(1)$

Thus, $f(n) = \Omega(n) = \Omega(1)$
The prescribed lower bound for $f(n)$ should be $\Omega(n)$ as the it is the tightly lower bound time-complexity of the function.

(c)
- $f(n) = n^2$
  $f(n) \leq n^2$
  Here, $c_1 = 1$ $n_0 = 0$

  Also,
  $f(n) = n^2$
  $f(n) \geq n^2$
  Here, $c_2 = 1$ $n_0 = 1$
  $\therefore f(n) = \theta(n^2)$

- $g(n) = 2n^2$
  $g(n) \leq 2n^2$
  Here, $c_1 = 2$ $n_0 = 0$

Also,
$g(n) = 2n^2$
$g(n) \geq n^2$
Here, $c_2 = 1$ $n_0 = 1$
$\therefore g(n) = \theta(n^2)$

(d) Let $f(n)$ and $g(n)$ be any two functions
Assume that $f(n) = \theta(g(n))$
Therefore, by definition,
$c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\quad$ c_1, c_2 are constants

- Consider
$f(n) \leq c_2 g(n)$
$\therefore f(n) = O(g(n))$

- Consider
$c_1 g(n) \leq f(n)$
$\therefore f(n) = \Omega(g(n))$

Conversely, if given
$f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$
then, by definition,
$f(n) \geq c_1 g(n)$ and $f(n) \leq c_2 g(n)$ $\quad$ c_1, c_2 are constants
$\therefore c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\therefore f(n) = \theta(n)$

(e) (i) $f(n) = 1 + 2 + 3 + \dots n$
$f(n) \leq n + n + n + \dots n$
$f(n) \leq n * n$
$f(n) \leq n^2$
$f(n) \leq c_1 n^2$ $\quad$ c_1 is constant $\quad$ ....(A)

$f(n) = 1 + 2 + 3 + \dots n$
$f(n) \geq n/2 + n/2 + n/2 + \dots n/2$
$f(n) \geq n * n/2$
$f(n) \geq n^2/2$
$f(n) \geq c_2 n^2$ $\quad$ c_2 is constant $\quad$ ....(B)

From (A) and (B)
$c_1 n^2 \leq f(n) \leq c_2 n^2$
$\therefore \boldsymbol{f(n) = \theta(n)}$

(ii) $f(n) = 2n^3 - n^2$
$f(n) \leq 2n^3$
Here, $c = 2$ and $g(n) = n^3$ and $n_0 = 1$
$\therefore \boldsymbol{f(n) = O(n^3)}$

(iii) $f(n) = 7n^2 logn + 25000n$
$f(n) \leq 7n^2 logn + 25000n^2 logn$
Here, $c = 25007$ and $g(n) = n^2 logn$ and

$\therefore f(n) = O(n^2 logn)$

(f) Given: $T1(n) = O(f(n))$ and $T2(n) = O(g(n))$

(a) To prove: $T1(n) + T2(n) = max(O(g(n), O(f(n))$

$T1(n) + T2(n) = O(f(n)) + O(g(n))$

if $O(f(n)) > O(g(n))$,
$T1(n) + T2(n) \leq O(f(n)) + O(f(n))$
$T1(n) + T2(n) \leq 2O(f(n))$
$T1(n) + T2(n) = O(f(n))$

if $O(f(n)) < O(g(n))$,
$T1(n) + T2(n) \leq O(g(n)) + O(g(n))$
$T1(n) + T2(n) \leq 2O(g(n))$
$T1(n) + T2(n) = O(g(n))$

$\therefore T1(n) + T2(n) = max(O(g(n), O(f(n))$

(b) To prove: $T1(n) * T2(n) = O((g(n) * (f(n))$

$T1(n) = O(f(n))$
$\therefore T1(n) \leq c_1 f(n)$ c_1 is constant

$T2(n) = O(g(n))$
$\therefore T2(n) \leq c_2 g(n)$ c_2 is constant

$\therefore T1(n) * T2(n) \leq c_1 f(n) * c_2 g(n)$
$\therefore T1(n) * T2(n) \leq (c_1 * c2) * f(n) * c_2 g(n)$
$\therefore T1(n) * T2(n) \leq c_3 * f(n) * g(n)$ ...$c_3 = c_1 * c_2$ (constant)
$\therefore T1(n) * T2(n) = O((g(n) * (f(n))$

(g) $f(n) \leq f(n) + g(n)$
Also,$g(n) \leq f(n) + g(n)$
$\therefore max(f(n), g(n)) = O(f(n) + g(n))$ ...(A)

Similarly,
$max(f(n), g(n)) \geq 1/2(f(n) + g(n))$
$\therefore max(f(n), g(n)) = \Omega(f(n) + g(n))$ ...(B)

From (A) and (B)
$max(f(n), g(n)) = \theta(f(n) + g(n))$

(h) (a) $f(n) = n^2 2^n + n^{100}$
$f(n) \leq n^2 2^n + n^2 2^n$
$f(n) \leq 2n^2 2^n$
Here, $c_1 = 2$ and $g(n) = n^2 2^n$
Also,
$f(n) \geq n^2 2^n$
Here, $c_2 = 1$ and $g(n) = n^2 2^n$
$\therefore c_2 g(n) \leq f(n) \leq c_1 g(n)$
$\therefore f(n) = \theta(n^2 2^n)$ .... By definition

(b) $f(n) = n^2/logn$
$f(n) \leq n^2$
$\therefore f(n) = O(n^2)$
However, $f(n) \neq \Omega(n^2)$
Now, For any two functions $f(n)$ and $g(n)$, $f(n) = \theta(g(n))$ only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
$\therefore f(n) \neq \theta(n^2)$

(i) Given: $T(x)$ is a polynomial of degree n
Let $T(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots a_mx^n$
$\therefore T(x) \leq a_0x^n + a_1x^n + a_2x^n + a_3x^n + \ldots a_mx^n$
$\therefore T(x) \leq (a_0 + a_1 + a_2 + a_3 + \ldots a_m)x^n$
Let $(a_0 + a_1 + a_2 + a_3 + \ldots a_m) = c_1$
$\therefore T(x) \leq c_1x^n$ ...(A)

$T(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots a_mx^n$
$\therefore T(x) \geq a_mx^n$
Let $a_m = c_1$
$\therefore T(x) \geq c_2x^n$ ...(B)

From (A) and (B)
$c_2(x^n) \leq T(x) \leq c_1(x^n)$
By definition,
$T(x) = \theta(x^n)$

(j) $P(n) = a_0 + a_1n + a_2n^2 + \ldots a_mn^m$
$\therefore P(n) \leq a_0n^m + a_1n^m + a_2n^m + a_3n^m + \ldots a_mx^m$
$\therefore P(n) \leq (a_0 + a_1 + a_2 + a_3 + \ldots a_m)x^m$
Let $(a_0 + a_1 + a_2 + a_3 + \ldots a_m) = c_1$
$\therefore P(n) \leq c_1n^m$
$\therefore P(n) = O(n^m)$

(k) Let, T(n) be the running time complexity.
Assume, n = 4

$i = 1$	$j = 1$	$k = 1$
$i = 2$	$j = (1), (2)$	$k = (1), (1, 2)$
$i = 3$	$j = (1), (2), (3)$	$k = (1), (1, 2), (1, 2, 3)$
$i = 4$	$j = (1), (2), (3), (4)$	$k = (1), (1, 2), (1, 2, 3), (1, 2, 3, 4)$

$\therefore \frac{n(n+1)(2n+1)}{6}$

$\therefore T(n) = O(n^3)$

(l) (i) $t_{A(n)} = 1000n$ and $t_{B(n)} = 10n^2$
if $t_{A(n)}$ is faster than $t_{B(n)}$
$t_{A(n)} > t_{B(n)}$
$\therefore 1000n > 10n^2$

$\therefore 1000n - 10n^2 > 0$
$\therefore 10n(100 - n) > 0$
$\therefore n > 0$ or $100 > n$
$\therefore n \in (0, 100)$

(ii) $t_{A(n)} = 1000nlog_2 n$ and $t_{B(n)} = n^2$
$t_{A(n)} > t_{B(n)}$
$\therefore 1000nlog_2 n > n^2$
$\therefore 1000nlog_2 n - n^2 > 0$
$\therefore n(1000log_2 n - n) > 0$
$\therefore n > 0$ or $1000log_2 n > n$

(iii) $t_{A(n)} = 2n^2 and t_{B(n)} = n^3$
$t_{A(n)} > t_{B(n)}$
$\therefore 2n^2 > n^3$
$\therefore 2n^2 - n^3 > 0$
$\therefore n^2(2 - n) > 0$
$\therefore n^2 > 0$ or $2 > n$
$\therefore n \in (0, 2)$

(iv) $t_{A(n)} = 2n and t_{A(n)} = 100n$
$t_{A(n)} > t_{B(n)}$
$\therefore 2n > 100n$
$\therefore 2n - 100n > 0$
$\therefore -98n > 0$
$\therefore n \in (-\infty, 0)$

(m) Consider an input array A of n elements. Each element is an n-bit integer except 0. In this scenario, I recommend using the Radix Sort algorithm for sorting the array. Here's why:

(a) **Stable Sorting**: Radix Sort is a stable sorting algorithm, which means it preserves the relative order of equal elements. This is important when you want to maintain any existing order in your data.

(b) **Linear Time Complexity**: Radix Sort has a time complexity of $O(nk)$, where n is the number of elements, and k is the number of digits in the largest number. In this case, the largest number has n bits, so k is also n. This results in a linear time complexity of $O(n \cdot n)$, which simplifies to $O(n)$.

(c) **No Comparison Operations**: Unlike comparison-based sorting algorithms (e.g., QuickSort, MergeSort), Radix Sort does not require comparing elements to each other. Instead, it distributes elements into buckets based on each digit's value (in base-n), and this process is done for each digit. This makes it efficient for large data sets.

(d) **Predictable Performance**: The performance of Radix Sort is predictable and does not depend on the specific input distribution. It works well for both uniformly distributed and non-uniformly distributed data.

(e) **In-Place or Out-of-Place**: Radix Sort can be implemented in an in-place manner or with additional memory for intermediate data structures, depending on your memory constraints.

(f) **Efficient for Large Integers**: Since your input consists of n-bit integers, Radix Sort is efficient because it takes advantage of the fixed size of the integers and doesn't rely on complex comparison operations.

In summary, Radix Sort is a great choice for sorting an array of n-bit integers, including cases where the largest integer can be represented using n bits. It offers predictable performance with a time complexity of $O(n)$ and is efficient for large integers.

(n) Given: Input array a[1..n] of arbitrary numbers

$O(1)$ implies that the number of distinct elements are independent of the size of the array. The number of distinct elements remain constant even if the size of array (n) changes .