1 Solutions

Question 1 ():

Solution:

(a) $O(2^n)$. *The last term, which is exponential, is the fastest growing term.*

(b) $O(n^4)$. *The last term simplifies to just n, so the n^4 term grows fastest. Note: $n^2 \cdot \log n^2 = 2n^2 \log n < n^2 \cdot n = n^3$, so the middle term does not grow as fast as n^3.*

(c) $O(n \log_2 n)$. *The middle term $29n \log_2 n$ is $O(n \log_2 n)$ which grows more quickly than $O(n)$ which grows more quickly than $n^{0.7}$.*

Question 2 ():

Solution: Answers for each part:

a) Answer: Statements 3 and 4 are true. *Explanation: Statement 3 is true because n^3 is the tightest upper bound ($19n^3$) is the fastest growing term, and stripping the constants leaves n^3. But 2^n grows even more quickly than n^3 so if n^3 is an upper bound, then 2^n must also be an upper bound.*

c) Answer: $\Theta(n^3)$. *Explanation: Since the third term in the expression is inarguably $\Theta(n^3)$ and it is the most quickly growing thrm, then $T_A(n) \in \Theta(n^3)$. This is the only possible answer because big-Θ implies that the given function is not only an upper bound, but also a lower bound. Anything that grows more quickly that n^3 would not be a lower bound.*

Question 3 ():

Solution:

(a) $O(n^2)$. *By first rule in lecture 4, slide 11.*

(b) $O(n^2 2^n)$, or equivalently $O(2^n n^2)$. *By second rule in lecture 4, slide 11.*

(c) $O(n^3)$. *By first applying rule 3 to each term, then applying rule 1.*

(d) $O(n^2 \log_2 n^2 + m)$ *By rule #1 in lecture 4, slide 11. This is as much as we can simplify because we can't possibly know which term grows faster because m and n are independent.*

Question 4 ():

Solution: This is a classic quadratic loop. The inner loop executes n times for values of j from 1 to n. Each time through the loop, 2 statements are executed (the loop condition and the print statement). This is a total of $2n$ statements. There is an extra statement for when the loop condition becomes false. So the total number of statements for the inner loop is $2n + 1$. The outer loop executes n times for values of i between 1 and n. For each of those values of i, we execute the $2n + 1$ statements for the inner loop, plus an additional statement for the outer loop condition, or $2n + 2$ statements per iteration of the outer loop. There are n iterations of the outer loop, so this totals to $n(2n + 2) + 1$ where the extra $+1$ is for the outer loop condition being false. Thus the number of statements executed is exactly $2n^2 + 2n + 1$ or $\Theta(n^2)$.

Question 5 ():

Solution:

(a) This is a *dependent quadratic loop* where the number of iterations of the inner loop depends on the value of i in the outer loop.

The inner loop executes two statements per loop iteration (the print statement and the loop condition). The inner loop condition is true $n-i-1$ times, depending on the value of i. So the total number of statements executed by the inner loop is $2(n-i-1)+1$

(since this is pseudocode, there is no explicit loop condition to count, so if the loop condition was not included, that solution is also acceptable, in which case the number of statements executed by the inner loop is $(n-i-1)+1$ and the factor of 2 disappears from the simplification below; it's also not a big deal if they miss the $+1$ for the loop condition being false, in which case the trailing n term disappears from the simplification below).

The outer loop executes n times, for values of i from 0 to $n-1$. Thus the total number of statements executed is the sum of the number of statements executed by the inner loop for each of these values of i:

$$\begin{aligned}
\sum_{i=0}^{n-1}[2(n-i-1)+1] &= 2\times((n-1)+(n-2)+(n-3)+\cdots+2+1+0)+n \\
&= 2\times(0+1+2+\cdots+(n-3)+(n-2)+(n-1))+n \\
&= 2\sum_{i=0}^{n-1} i+n \quad \textit{(a familiar sum for which we know a closed form!)} \\
&= 2(n-1)(n)/2+n \\
&= n^2
\end{aligned}$$

(b) $\Theta(n^2)$

Question 6 ():

Solution: The active operation is the inner-loop condition. It is executed one more time than the print statement and all individual statements are $O(1)$.

The cost of the active operation is $O(1)$. The active operation is executed $(n-i-1)+1$ times by the inner loop. The outer loop executes n times, once each for the values of i from 0 to $n-1$. Thus the total cost of all active operation invocations is:

$$\begin{aligned}
&\sum_{i=0}^{n-1}[O(1)\cdot((n-i-1)+1)]\\
&\sum_{i=0}^{n-1}[(n-i-1)+1]\\
&=\sum_{i=0}^{n-1}\cdot(n-i-1)+n\\
&=\sum_{i=0}^{n-1}i+n \quad \textit{(By same reasoning as previous question)}\\
&=(n)(n+1)/2+n\\
&=n^2/2+3n/2
\end{aligned}$$

Question 7 ():

Solution: The active operation must be the first item of the loop body because even though it executes one less time than the loop condition, the call to `binarySearch` is not constant time. All other method calls are constant-time.

The cost of the active operation is $O(\log m)$.

The worst case is when the target is not found in any of the arrays in which case all n arrays are searched. This means that the active operation executes n times.

The total cost of all active operation invocations is: $n\cdot\log m$ which is $O(n\log m)$.

Question 8 ():

Solution:

Name: PriorityQueue$<G>$

Sets: Q : set of priority queues containing elements from G.

G : set of items that can be in a priority queue.

B : $\{true, false\}$

$\mathbb{N}$: set of positive integers.

$\mathbb{N}_0$: set of non-negative integers.

Signatures: newPriorityQueue$<G>$: $\mathbb{N} \to Q$

Q.insert(g): $G \not\to Q$

Q.isFull: $\to B$

Q.isEmpty: $\to B$

Q.count: $\to \mathbb{N}_0$

Q.maxItem: $\not\to G$

Q.minItem: $\not\to G$

Q.deleteMax: $\not\to Q$

Q.deleteMin: $\not\to Q$

Q.frequency(g): $G \to N_0$

Q.deleteAllMax: $\not\to Q$

Preconditions: For all $q \in Q, g \in G$,

q.insert(g): queue is not full

q.maxItem: queue is not empty

q.minItem: queue is not empty

q.deleteMax: queue is not empty

q.deleteMin: queue is not empty

q.deleteAllMax: q must not be empty.

(Operations without preconditions are omitted)

Semantics: For all $n \in N, g \in G, n \in \mathbb{N}$

newPriorityQueue<G>(n): create a new queue with capacity n.

q.insert(g): insert item g into t in priority order with the highest number being the highest priority.

q.isFull: return $true$ if t is full, $false$ otherwise

q.isEmpty: return $true$ if t is empty, $false$ otherwise

q.count: obtain number of items in q

q.maxItem: return the largest (highest priority) item in q.

q.minItem: return the smallest (lowest priority) item in q.

q.deleteMax: remove the largest (highest priority) item in q from q.

q.deleteMin: remove the smallest (lowest priority) item in q from q.

q.frequency(g): return number of times element g appears in the queue regardless of priority.

q.deleteAllMax: all occurrences of the highest priority item are deleted from q.