Recitation 3: Big-O and Computational Complexity

1 Big-O Notation

True or False? Give a brief justification for each.

1. $n \log n \in O(n)$
2. $10^9 n \in \Theta(n)$
3. $\log(n^4) = O(\log n)$
4. $n^6 \cdot 2^{-n} \in O(1)$
5. $\log n \in \Omega(\sqrt{n})$.
6. $2^{\sqrt{n}} \in O(n^3)$
7. $3^n \in \Omega(n!)$
8. $n^k \in O(k^n)$ for $k \geq 2$

Solutions

1. **False.** For large n, $\log n > 1$, hence $n \log n > n$.
2. **True.** Multiplying by a constant preserves $\Theta(\cdot)$.
3. **True.** $\log(n^4) = 4 \log n = O(\log n)$.
4. **True.** $n^6/2^n \to 0$; in particular it is bounded, so $O(1)$.
5. **False.** In fact, $\log n = O(n^\epsilon)$ for every $\epsilon > 0$ (see proof below).
6. **False.** Take logs: compare $\sqrt{n}$ vs $3 \log n$; $\sqrt{n}$ dominates, so $2^{\sqrt{n}}$ eventually outgrows n^3.
7. **False.** Stirling: $n! \sim \sqrt{2\pi n}(n/e)^n \gg 3^n$, so $3^n = o(n!)$ (not Ω).
8. **True** Exponential k^n dominates any polynomial n^k. Take logs: $\log(n^k) = k \log(n) \leq n \log(k) = \log(k^n)$.

Formally prove or disprove the following statements:

1. If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
2. $n^2 + 3/n \in O(n^2)$
3. $n^2 + 3n \log n \in O(n^2)$.

4. $n \log n \in O(n^{1+\epsilon})$ for every $\epsilon > 0$.

Solutions.

1. **Proof.** $f \in O(g) \Rightarrow \exists c_1, n_1$ s.t. $f(n) \leq c_1 g(n)$ for $n \geq n_1$. Also $g \in O(h) \Rightarrow \exists c_2, n_2$ s.t. $g(n) \leq c_2 h(n)$ for $n \geq n_2$. Then for $n \geq \max\{n_1, n_2\}$, $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$, so $f \in O(h)$.

2. **Proof.** For $n \geq 1$, $3/n \leq 3 \leq 3n^2$. Hence $n^2 + 3/n \leq n^2 + 3 \leq 4n^2$. Take $c = 4, n_0 = 1$.

3. **Proof.** For $n \geq 2$, $\log n \leq n$, so $3n \log n \leq 3n^2$. Thus $n^2 + 3n \log n \leq 4n^2$ for $n \geq 2$; take $c = 4, n_0 = 2$.

4. **Proof.** Fix $\epsilon > 0$. Note that $\lim_{n\to\infty} \frac{\log n}{n^\epsilon} = 0$, so $\exists n_0$ s.t. $\log n \leq n^\epsilon$ for $n \geq n_0$. Then $n \log n \leq n \cdot n^\epsilon = n^{1+\epsilon}$ for $n \geq n_0$, i.e. $n \log n \in O(n^{1+\epsilon})$.

2 Analyzing runtime complexity

1. Analyze the running time of the following program:

   $v = 0$
   for $i = 1$ to n **do**
       for $j = 1$ to n **do**
           $v = i \cdot j^2$
       end for
   end for

   Solution. The body runs once per pair (i, j): exactly n^2 executions of an $O(1)$ statement. Total time $\Theta(n^2)$; total body executions $= n^2$.

2. Suppose the function $F(i, j)$ takes $\Theta(i)$ time to execute. Analyze the runtime of the following program, which has input n:

   for $i = 1$ to n **do**
       for $j = i$ to n **do**
           Call $F(i, j)$
       end for
   end for

   Hint: $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6$.

   Solution. Let $T(n)$ be the running time of the algorithm. Since $F(i, j) = \Theta(i)$, summing over the two loops gives $T(n) = \sum_{i=1}^n \sum_{j=i}^n \Theta(i) = \sum_{i=1}^n (n - i + 1)\Theta(i) = \Theta\big(\big(\sum_{i=1}^n i(n - i + 1)\big)$. Compute

   $$\sum_{i=1}^n i(n+1) - i^2 = (n+1)\frac{n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} = \frac{n(n+1)(n+2)}{6} = \Theta(n^3).$$

   Hence the runtime is $\Theta(n^3)$.

3. Analyze the runtime of the following binary search algorithm. Assume that it is first called as BinSearch(A, 0, n, x). Here A is the (sorted) array to be searched and the goal is to return the index of x of it's found in A, and -1 otherwise.

   procedure BinSearch(A, ℓ, r, x)

```
    if ℓ > r then
        return -1
    end if
    m ← ℓ + ⌊(r − ℓ)/2⌋
    if A[m] = x then
        return m
    else if A[m] < x then
        return BinSearch(A, m + 1, r, x)
    else
        return BinSearch(A, ℓ, m − 1, x)
    end if
end procedure
```

Solution. Let $T(n)$ be the worst-case time on an interval of length $n = r - \ell + 1$. Then

$$T(n) = T(\lfloor n/2 \rfloor) + O(1), \qquad T(1) = O(1).$$

Recursively unravel this equation:

$$T(n) = T(\lfloor n/2 \rfloor) + O(1) = T(\lfloor n/4 \rfloor) + 2 \cdot O(1) = \cdots = T(\lfloor n/2^k \rfloor) + kO(1).$$

Choose $k = \lceil \log_2 n \rceil$, so $\lfloor n/2^k \rfloor \le 1$ and $T(n) \le T(1) + \lceil \log_2 n \rceil \cdot O(1) = O(\log n)$.

4. Analyze the runtime of the following sorting algorithm. Assume the $\textsc{Merge}(A_1, A_2)$ procedure takes time $O(n_1 + n_2)$ where A_1 has length n_1 and A_2 has length n_2.

```
procedure MergeSort(A, ℓ, r)
    if ℓ ≥ r then
        return
    end if
    m ← ℓ + ⌊(r − ℓ)/2⌋
    A1 ← MergeSort(A, ℓ, m)
    A2 ← MergeSort(A, m + 1, r)
    Merge(A1, A2)
end procedure
```

Solution. Let $T(n)$ be the worst-case time on an array of length n. For a merge that costs $O(n)$ we have

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n), \qquad T(1) = O(1).$$

Recursively unravel this equation (upper-bounding $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) \le 2\,T(\lceil n/2 \rceil)$):

$$\begin{aligned} T(n) &\le 2\,T(\lceil n/2 \rceil) + O(n) \\ &\le 2^2\,T\left(\left\lceil \frac{n}{2^2} \right\rceil\right) + 2 \cdot O(n) \\ &\;\vdots \\ &\le 2^k\,T\left(\left\lceil \frac{n}{2^k} \right\rceil\right) + k\,O(n). \end{aligned}$$

Choose $k = \lceil \log_2 n \rceil$, so $\lceil n/2^k \rceil \le 1$ and hence $T(\lceil n/2^k \rceil) = O(1)$. Therefore

$$T(n) \le 2^k\,O(1) + k\,O(n) = O(n) + O(n \log n) = O(n \log n).$$

If we want a lower bound we can argue as follows: At each recursion level the two merges touch a total of n elements, so the work per level is $\Omega(n)$, and there are $\lfloor \log_2 n \rfloor + 1$ levels; thus $T(n) = \Omega(n \log n)$. Combining gives $T(n) = \Theta(n \log n)$.