Algorithmic Analysis

Problem 1

Consider the following program with two unspecified lines.

$$\begin{array}{l}
\texttt{for } j = 1 \texttt{ to } n: \\
\quad (*) \\
\quad \texttt{while } i > 1: \\
\quad\quad \texttt{print } i \\
\quad\quad (**) \\
\quad \texttt{end while} \\
\texttt{end for}
\end{array}$$

Give an asymptotic upper bound on the running time, in terms of n for the given program when the missing lines are specified as follows:

(a) $(*) : i = n \qquad (**) : i = i - 1$

(b) $(*) : i = n \qquad (**) : i = i/2$

(c) $(*) : i = j \qquad (**) : i = i - 2$

(d) $(*) : i = j \qquad (**) : i = i/2$

Solution

The for-loop will execute $O(n)$ times, the choice of $(*)$ and $(**)$ determine how many times the inner while-loop will execute. The innermost code takes $O(1)$ time to execute, as does every other line not associated with a loop. So in all cases, the running time will be $O(1) \times O(n) = O(n)$ times the number of executions of the inner while-loop.

(a) In this case the while-loop executes $O(n)$ times for each iteration of the for-loop, so the running time is bounded above by $O(n) \times O(n) = O(n^2)$.

(b) In this case the while-loop executes $O(\log n)$ times, so the running time is bounded above by $O(n) \times O(\log n) = O(n \log n)$.

(c) In this case the number of executions of the while-loop changes with each iteration of the for-loop: the while-loop executes $j/2 = O(j)$ times in each iteration. Since $j \leq n$ we could use $O(n)$ as an upper bound for the number of executions of the while-loop in each iteration of the for loop, giving us a running time of $O(n^2)$ as with (a). However, it may be possible to obtain a better upper bound by summing the for-loop executions individually. This would give us a total running time of $O(1) + O(2) + \ldots + O(n)$, but this is also $O(n^2)$.

(d) In this case the while-loop executes $O(\log j)$ times. Again, we could use the fact that $j \leq n$ to simplify, giving an upper bound of $O(\log n)$ iterations of the while loop, and an overall running time of $O(n \log n)$ as with (b). Can we do better by summing the for-loop executions individually? We observe that for $j \in [n/2, n]$, $\log j \in [\log n - 1, \log n]$, so at least $n/2$ executions of the for-loop will take $\Omega(\log n)$ time. Therefore $O(n \log n)$ is the best upper bound we can obtain.

Problem 2
Analyse the complexity of the following algorithms to compute the n-th Fibonacci number

(a) **FibOne**(n):

if $n \leq 2$ then return 1

else return **FibOne**$(n-1)$ + **FibOne**$(n-2)$

(b) **FibTwo**(n):

$x = 1, y = 0, i = 1$

While $i < n$:

$t = x$

$x = x + y$

$y = t$

$i = i + 1$

return x

Solution

(a) Let $T(n)$ be the running time of **FibOne**(n). Then in the worst case, there are two recursive calls to smaller instances of **FibOne**, taking time $T(n-1)$ and $T(n-2)$ respectively. All other operations are constant time, so

$$\begin{aligned} T(n) &= O(1) + T(n-1) + T(n-2) \\ &\leq O(1) + 2.T(n-1). \end{aligned}$$

From the lectures, this means that $T(n) \in O(2^n)$.

(b) Let $T(n)$ be the running time of **FibTwo**(n). We have a while-loop which runs $O(n)$ times, and within the while loop there are several operations taking $O(1)$ time. All other operations are constant time, so the overall running time is $O(1) + O(n) \times O(1) = O(n)$.

Discussion

NB: It is possible to obtain better bounds for **FibOne**, however because of the $O(1)$ that appears in the recurrence equation, it is not quite as simple as $T(n) = \textsc{Fib}(n)$. A bound of $O(2^n)$ demonstrates a reasonable level of understanding, so would be sufficient in most assessable tasks.

Problem 3
Analyse the complexity of the following recursive algorithm to test whether a number x occurs in an *ordered* list $L = [x_1, x_2, \ldots, x_n]$ of size n. Take the cost to be the number of list element comparison operations.

BinarySearch$(x, L = [x_1, x_2, \ldots, x_n])$:

if $n = 0$ then return no

else

if $x_{\lceil \frac{n}{2} \rceil} > x$ then return **BinarySearch**$(x, [x_1, \ldots, x_{\lceil \frac{n}{2} \rceil - 1}])$

else if $x_{\lceil \frac{n}{2} \rceil} < x$ return **BinarySearch**$(x, [x_{\lceil \frac{n}{2} \rceil + 1}, \ldots, x_n])$

else return yes

Solution

Let $T(n)$ be the cost of running **BinarySearch** on a list of length n. In the worst case, we make $2 = O(1)$ element comparisons and recursively call **BinarySearch** on a list of length $\lceil \frac{n}{2} \rceil$. So we have:

$$T(n) = O(1) + T(n/2).$$

The Master Theorem applies to this recurrence: we have $a = 1$, $b = 2$, $c = 0$ and $d = \log_b(a) = 0$, so we are in Case 2. This tells us that $T(n) \in O(n^d \log n) = O(\log n)$.