CS 5800 Assignment 1 Solutions

Course Staff

July 20, 2017

1 Big-O and friends

(a) Yes. Since $2^{n+1} = 2 \cdot 2^n$, we choose (for example) $c = 2$ and $N = 1$.

(b) *Proof by induction on n.* Base case $n = 10$.

$$2^n = 1024 \geq 1000 = n^3$$

For the inductive step, we assume the proposition true for $n = k$, that is $2^k \geq k^3$, and show for $n = k + 1$. Working backwards,

$$\begin{aligned}
(k+1)^3 &= k^3 + 3k^2 + 3k + 1 \\
&\leq k^3 + 3k^2 + 3k^2 + k^2 && \text{because } k \geq 1 \\
&\leq k^3 + k^3 && \text{because } k \geq 7 \\
&\leq 2^k + 2^k && \text{by our induction hypothesis} \\
&\leq 2^{k+1} && \square
\end{aligned}$$

Finally, we assert that $2^n = \Omega(n^3)$ by choosing $c = 1$ and $N = 10$.

(c) *Proof.* Consider the limit,

$$\lim_{n\to\infty} \frac{(n+a)^b}{n^b} = \lim_{n\to\infty} \left(\frac{n+a}{n}\right)^b = 1$$

Using the definition for a limit going to infinity, and taking (for example) $\varepsilon = 1/2$, there exists an N such that for all $n > N$

$$\left| \frac{(n+a)^b}{n^b} - 1 \right| < \frac{1}{2}$$

Or,

$$\frac{1}{2} n^b < (n+a)^b < \frac{3}{2} n^b$$

So to satisfy the definition for big-Θ, we choose $c_1 = 1/2$, $c_2 = 3/2$, and N as above. $\square$

(d) This is true. Observe that $\max\{f(n), g(n)\} \leq f(n) + g(n)$, and also that $2 \cdot \max\{f(n), g(n)\} \geq f(n) + g(n)$. So to meet the definition of big-Θ, we choose $c_1 = 1/2$, $c_2 = 1$, and $N = 1$.

(e) This is also true.

Proof. ($\Rightarrow$) Since $f(n) = \Theta(g(n))$, we have c_1, c_2, and N that satisfy the definition. For $f(n) = O(g(n))$, we choose $c = c_2$. For $f(n) = \Omega(g(n))$, we choose $c = c_1$.

($\Leftarrow$) Since $f(n) = O(g(n))$, we have constants c_O and N_O that satisfy the definition. Likewise, for $f(n) = \Omega(g(n))$, we have constants c_Ω and N_Ω. To show $f(n) = \Theta(g(n))$, we choose $c_1 = c_\Omega$, $c_2 = c_O$, and $N = \max\{N_\Omega, N_O\}$. $\square$

(f) This is false. In particular, the $\Leftarrow$ implication does not hold. For example, let $f(n) = 2^{n \cdot \sin(n)}$ and $g(n) = 2^n$. Since the sine function has a maximum value of one, $f(n) = O(g(n))$, and since $f(n)$ can take on arbitrarily small positive values, $f(n) \neq \Theta(g(n))$. However, the local maxima of $f(n)$ grow at the same rate as $g(n)$, so $f(n) \neq o(g(n))$.

(If one requires that f be monotonic, we might imagine a step function where the "upper corners" of each step are along $y = n$ and the "lower corners" are along $y = \log n$. Then set $g(n) = n$, and we have another counterexample.)

2 Prove the recurrence

To show that $T(n) = O(n)$ we need to show that there exist constants $c > 0$ and $n_0 > 0$ such that $T(n) < c \cdot n$ for any $n > n_0$.

The key point here is to come up with stronger inductive hypothesis than simply $T(n) < c \cdot n$.

Using the induction hypothesis:

$T(k) \leq 2k - 1$ for $k < n$, we get (note that both $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ are smaller than n for any $n > 1$) :

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \leq 2\lfloor n/2 \rfloor - 1 + 2\lceil n/2 \rceil - 1 + 1 = 2n - 1$

Since the base case holds (indeed $T(1) = 1 \leq 2 \cdot 1 - 1 = 1$), It follows that for any n we have:

$T(n) \leq 2n - 1 < 2n$

And hence $T(n) = O(n)$.

3 Master theorem

We employ the Master Theorem for standard recurrence equations:

$T(n) = aT(n/b) + f(n)$

Where n/b can be either $\lfloor n/b \rfloor or \lceil n/b \rceil$.

(a) $T(n) = 4T(\lfloor n/2 \rfloor) + 7n$

where $a = 4, b = 2, f(n) = 7n$ so $n^{\log_b a} = n^2$

$f(n) = O(n^{\log_b a - \epsilon})$. Hence applying the Master Theorem

$T(n) = \theta(n^2)$

(b) $T(n) = 4T(\lfloor n/2 \rfloor) + 5n^2$

where $a = 4, b = 2, f(n) = 5n^2$ so $n^{\log_b a} = n^2$

$f(n) = \theta(n^{\log_b a})$. Hence applying the Master Theorem

$T(n) = \theta(n^2 logn + n^2)$

$= \theta(n^2 logn)$

(c) $T(n) = 4T(\lfloor n/2 \rfloor) + 12n^3$

where $a = 4, b = 2, f(n) = 12n^3$ so $n^{\log_b a} = n^2$

$f(n) = \Omega(n^{\log_b a + \epsilon})$ and additionally

$af(n/b) \leq cf(n)$ i.e. $4 \cdot 12(n/2)^3 \leq c \cdot 12n^3$ for some constant c.

Hence applying the Master Theorem

$T(n) = \theta(f(n))$

$= \theta(n^3)$

4 Binary search

4.a Pseudocode for BinarySearch

If a matching element is not found, the index of the largest element less than the given value is returned. If there is no element smaller than the given value, $start - 1$ is returned.

```
procedure BinarySearch($A, v, start, end$)
    while $start \leq end$ do
        $mid \leftarrow \left\lfloor \frac{start + end}{2} \right\rfloor$
        if $A[mid] < v$ then
            $start \leftarrow mid + 1$
        else if $A[mid] > v$ then
            $end \leftarrow mid - 1$
        else
            return $mid$
    return $end$
```

In this algorithm, our recurrence equation for total number of comparisons would be: $T(n) = T(n/2) + 3$, where $a = 1, b = 2, f(n) = 3$

This is an instance of Case 2 of the Master Theorem, because $f(n) = 3 = \theta(1)$ and $\theta(n^{\log_b a}) = \theta(n^{\log_2 1}) = \theta(1)$. So, $T(n) = \theta(n^{\log_2 1} \log_2 n) = \theta(\log_2 n)$.

4.b Improved Insertion Sort

The idea here is to improve the Insert operation by using binary search instead of linear search, to find the position of the next element to insert. We can use the BinarySearch procedure we defined above for that purpose.

```
procedure InsertionSort($A, n$)
    for $i = 1$ to $n$ do
        $Insert(A, A[i], i - 1)$
    return

procedure Insert(A, a, numFilled)
    $pos \leftarrow BinarySearch(A, a, 1, numFilled) + 1$
    $i \leftarrow numFilled$
    while $i > pos$ do
        $A[i] \leftarrow A[i-1]$
        $i \leftarrow i - 1$
    $A[pos] \leftarrow a$
    return
```

This algorithm performs a binary search n times, and thus it performs $O(n \log n)$ comparisons overall. Remember, this complexity does not count stores, etc.; it is primarily useful when comparisons are much more expensive than other operations.

5 Two-element sum

```
procedure Two-Element-Sum($A, x$)
    $sortedA \leftarrow MergeSort(A)$
    for $i = 1$ to $A.length$ do
        $r \leftarrow x - sortedA[i]$
        $j \leftarrow BinarySearch(sortedA, r, 1, A.length)$
        if $j > 0$ and $sortedA[j] = r$ and $i \neq j$ then
            return true
    return false
```

This procedure determines whether the given sum can be made from two of n elements, using $O(n \log n)$ basic operations.

6 Two smallest elements

Let us imagine a k-round tournament among the 2^k array elements. We consider each comparison as a match, in which the smaller element wins. Thus the winner of the tournament is the minimum element of the whole array. Observe that the second-minimum element must have lost to the winner in some match. If we trace the path of the winner through the tournament rounds, the second-minimum must be one of the k elements the winner defeated along the way.

It will be useful to us, then, to identify those elements that lost to the winner in the process of identifying the winner itself. We define a helper function that returns both the minimum element, and a list of possible second-minimums for a given section of the array. Only when we identify the final list of possible second-minimums do we perform comparisons along it.

```
1: **procedure** TwoSmallest($A, n$)
2:     $k \leftarrow \log_2 n$
3:     $m, L \leftarrow$ MinimumAndLosers($A, 1, n$)
4:     $m_2 \leftarrow L[1]$
5:     **for** $i = 2$ to $k$ **do**
6:         **if** $L[i] < m_2$ **then**
7:             $m_2 \leftarrow L[i]$
8:     **return** $m, m_2$

9: **procedure** MinimumAndLosers($A, s, t$)
10:     **if** $s = t$ **then**
11:         **return** $A[s], []$
12:     **else**
13:         $p \leftarrow \lfloor (s+t)/2 \rfloor$
14:         $m_\ell, L_\ell \leftarrow$ MinimumAndLosers($A, s, p$)
15:         $m_r, L_r \leftarrow$ MinimumAndLosers($A, p+1, t$)
16:         **if** $m_\ell < m_r$ **then**
17:             **return** $m_\ell, (m_r :: L_\ell)$
18:         **else**
19:             **return** $m_r, (m_\ell :: L_r)$
```

Our algorithm uses $n-1$ comparisons to simulate the tournament, as well as $k-1$ comparisons to identify the second-minimum from the list of losers. This gives a total of $n + \log_2 n - 2$ comparisons.