Data Structures and Algorithms
University of California, Irvine

January 24, 2019
EECS 114

Homework 1 Solutions

1 *(6 Points)* Asymptotic Notation

For each group of functions, sort the functions in increasing order of asymptotic (big-O) complexity:

1. *(2 Points)* **Group 1**

$$\begin{aligned} f_1(n) &= n^{0.999999} \lg n \\ f_2(n) &= 10000000n \\ f_3(n) &= 1.000001^n \\ f_4(n) &= n^2 \end{aligned}$$

Solution: The correct order of these functions is $f_1(n), f_2(n), f_4(n), f_3(n)$. To see why $f_1(n)$ grows asymptotically slower than $f_2(n)$, recall that for any $c > 0$, $\lg n$ is $O(n^c)$. Therefore we have:

$$f_1(n) = n^{0.999999} \lg n = O(n^{0.999999} n^{0.000001}) = O(n) = O(f2(n))$$

The function $f_2(n)$ is linear, while the function $f_4(n)$ is quadratic, so $f_2(n)$ is $O(f_4(n))$. Finally, we know that $f_3(n)$ is exponential, which grows much faster than quadratic, so $f_4(n)$ is $O(f_3(n))$.

2. *(2 Points)* **Group 2**

$$\begin{aligned} f_1(n) &= 2^{2^{1000000}} \\ f_2(n) &= 2^{100000n} \\ f_3(n) &= n \lg n \\ f_4(n) &= n\sqrt{n} \end{aligned}$$

Solution: The correct order of these functions is $f_1(n)$, $f_3(n)$, $f_4(n)$, $f_2(n)$. The variable n never appears in the formula for $f_1(n)$, so despite the multiple exponentials, $f_1(n)$ is constant. Hence, it is asymptotically smaller than $f_3(n)$, which does grow with n. We may rewrite the formula for $f_3(n)$ to be $f_3(n) = 2n \lg \sqrt{n} < 2n\sqrt{n}$. Hence, $f_3(n) = O(f_4(n))$. Finally, $f_2(n)$ is exponential, therefore, $f_4(n)$ is $O(f_2(n))$.

3. *(2 Points)* **Group 3**

$$
\begin{aligned}
f_1(n) &= n^{\sqrt{n}} \\
f_2(n) &= 2^n \\
f_3(n) &= n^{10}.2^{n/2} \\
f_4(n) &= \sum_{i=1}^{n} (i+1)
\end{aligned}
$$

Solution: The correct ordering of these functions is $f_4(n)$, $f_1(n)$, $f_3(n)$, $f_2(n)$. To see why, we first use the rules of arithmetic series to derive a simpler formula for $f_4(n)$:

$$f_4(n) = \sum_{i=1}^{n} (i+1) = \frac{n((n+1)+2)}{2} = \frac{n(n+3)}{2} = \Theta(n^2) \tag{1}$$

This is clearly asymptotically smaller than $f_1(n) = n^{\sqrt{n}}$. Next, we want to compare $f_1(n)$, $f_2(n)$, and $f_3(n)$. To do so, we transform both $f_1(n)$ and $f_3(n)$ so that they look more like $f_3(n)$:

$$
\begin{aligned}
f_1(n) &= n^{\sqrt{n}} = (2^{\lg n})^{\sqrt{n}} = 2^{\sqrt{n} \lg n} \\
f_3(n) &= n^{10}.2^{n/2} = 2^{\lg(n^{10})}.2^{n/2} = 2^{n/2 + 10 \lg n}
\end{aligned}
\tag{2}
$$

The exponent of the 2 in $f_1(n)$ is a function that grows more slowly than linear time; the exponent of the 2 in $f_3(n)$ is a function that grows linearly with n. Therefore, $f_1(n) = O(f_3(n))$. Finally, we wish to compare $f_3(n)$ with $f_2(n)$. Both have a linear function of n in their exponent, so its tempting to say that they behave the same asymptotically, but they do not. If c is any constant and $g(x)$ is a function, then $2^{cg(x)} = (2^c)^{g(x)}$. Hence, changing the constant of the function in the exponent is the same as changing the base of the exponent, which does affect the asymptotic running time. Hence, $f_3(n)$ is $O(f_2(n))$, but $f_2(n)$ is not $O(f_3(n))$.

2 *(20 Points)* Recurrences

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 10$. Make your bounds as tight as possible, and justify your answers.

(a) $T(n) = 2T(n/3) + n \lg n$

Solution: By Case 3 of the Master Method, we have $T(n) = \Theta(n \lg n)$.

(b) $T(n) = 3T(n/5) + \lg^2 n$

Solution: By Case 1 of the Master Method, we have $T(n) = \Theta(n^{\log_5(3)})$.

(c) $T(n) = T(n/2) + 2^n$

Solution: By Case 3 of Master Method, $T(n) = \Theta(2^n)$.

(d) $T(n) = T(\sqrt{n}) + \Theta(\lg \lg n)$

Solution: Change of variables: let $m = \lg n$. Recurrence becomes $S(m) = S(m/2) + \Theta(\lg m)$. Case 2 of Master Method applies, so $T(n) = \Theta((\lg \lg n)^2)$.

(e) $T(n) = 10T(n/3) + 17n^{1.2}$

Solution: Since $log_3 9 = 2$, so $log_3 10 > 2 > 1.2$. Case 1 of Master Method applies, $T(n) = \Theta(n^{log_3 10})$.

(f) $T(n) = 7T(n/2) + n^3$

Solution: By Case 3 of the Master Method, we have $T(n) = \Theta(n^3)$.

(g) $T(n) = T(n/2 + \sqrt{n}) + \sqrt{6046}$

Solution: By induction, $T(n)$ is a monotonically increasing function. Thus, for large enough n, $T(n/2) \leq T(n/2 + \sqrt{n}) \leq T(3n/4)$. At each stage, we incur constant cost $\sqrt{6046}$, but we decrease the problem size to atleast one half and at most three-quarters. Therefore $T(n) = \Theta(\lg n)$.

(h) $T(n) = T(n-2) + \lg n$

Solution: $T(n) = \Theta(n \lg n)$. This is $T(n) = \sum_{i=1}^{n/2} \lg 2i \geq \sum_{i=1}^{n/2} \lg i \geq (n/4)(\lg n/4) = \Omega(n \lg n)$. For the upper bound, note that $T(n) \leq S(n)$, where $S(n) = S(n-1) + \lg n$, which is clearly $O(nlgn)$.

(i) $T(n) = T(n/5) + T(4n/5) + \Theta(n)$

Solution: Master Method does not apply here. Draw recursion tree. At each level, do $\Theta(n)$ work. Number of levels is $log_{5/4} n = \Theta(\lg n)$, so guess $T(n) = \Theta(n \lg n)$ and use the substitution method to verify guess.

(j) $T(n) = \sqrt{n}T(\sqrt{n}) + 100n$

Solution: Master Method does not apply here directly. Pick $S(n) = T(n)/n$. The recurrence becomes $S(n) = S(\sqrt{n}) + 100$. The solution of this recurrence is $S(n) = \Theta(\lg \lg n)$. (You can do this using a recursion tree, or by substituting $m = \lg n$ again.) Therefore, $T(n) = \Theta(n \lg \lg n)$.

3 *(9 Points)* Sorting

1. *(9 Points)* Do Exercise 2.3-5 and 2.3-6 on page 39 in CLRS.

   Solution: 2.3-5

   Procedure Binary-Search takes in a sorted array A, a value v and a range $[low..high]$ of the array, in which we search for the value v. The procedure compares v to the array-element at the midpoint of the range and decides to eliminate half the range from further consideration. There are both iterative and recursive versions, each of which returns either an index i such that $[i] = v$, or NIL if no entry of $A[low..high]$ contains the value v. The initial call to either version should have the parameters $A, v, 1, n$ (index of the first array-element is 1).

Algorithm 1 Iterative binary search pseudo-code

```
1: procedure ITERATIVE-BINARY-SEARCH($A, v, low, high$)
2:     while $low \leq high$ do
3:         $mid = \lfloor (low + high)/2 \rfloor$
4:         if $v == A[mid]$ then
5:             return $mid$
6:         else if $v > A[mid]$ then
7:             $low = mid + 1$
8:         else
9:             $high = mid - 1$
10:    return NIL
```

Algorithm 2 Recursive binary search pseudo-code

```
1: procedure RECURSIVE-BINARY-SEARCH($A, v, low, high$)
2:     if $low > high$ then
3:         return $NIL$
4:     $mid = \lfloor (low + high)/2 \rfloor$
5:     if $v == A[mid]$ then
6:         return mid
7:     else if $v > A[mid]$ then
8:         return RECURSIVE-BINARY-SEARCH($A, v, mid + 1, high$)
9:     else
10:        return RECURSIVE-BINARY-SEARCH($A, v, low, mid - 1$)
```

Both procedures terminate the search unsuccessfully (returning NIL) when the range is empty (i.e. $low > high$) and terminate it successfully if the value v has been found. Based on the comparison of v to the middle element in the searched range, the search continues with the range halved. The recurrence for both these procedures is therefore $T(n) = T(n/2) + \theta(1)$, whose solution is $T(n) = \theta(lgn)$.

Solution: 2.3-6

The while loop of the Insertion-Sort procedure (lines 5-7 of the pseudo-code in the CLRS textbook) scans backward through the sorted array $A[i..j-1]$ to find the appropriate place for $A[j]$. The hitch is that the loop not only searches for the proper place for $A[j]$, but that it also moves each of the array elements that are bigger than $A[j]$ one position to the right (line 6). These movements can take as much as $\theta(j)$ time, which occurs when all the $j-1$ elements preceding $A[j]$ are larger than $A[j]$. We can use binary search to improve the running time of the search to $\theta(logj)$, but binary search will have no effect on the running time of moving elements. Therefore, binary search alone cannot improve the worst-case running time of Insertion-Sort to $\theta(nlogn)$.