# Algorithms and Data Structures
# MSIS 26:198:685
# Homework 1

**Instructor:** Farid Alizadeh
**Due Date:** Monday September 30, 2018, at 11:50PM

last updated on October 5, 2019

Please answer the following questions in **electronic** form and upload and submit your files to the Sakai site, before the due date. Make sure to push the **submit** button.

You can typeset your answer using MS Word (and MS equation for mathematical formulas), or LaTeX, or you may simply handwrite your note and scan and turn it not a **single pdf format file** and upload to Sakai.

1. An alternative definition for $\mathcal{O}$ notation is the following:

   > Consider two functions $\mathtt{f}(\mathtt{n})$ and $\mathtt{g}(\mathtt{n})$. We say that $\mathtt{f}(\mathtt{n}) \in \mathcal{O}_*\big(\mathtt{g}(\mathtt{n})\big)$ if for some constant $\mathtt{c}$ we have $\mathtt{f}(\mathtt{n}) \leq \mathtt{c}\mathtt{g}(\mathtt{n})$ *for infinitely many values of* $\mathtt{n}$.

   This definition is useful for cases where the performance of some algorithm may get better on some types of $\mathtt{n}$, even if they are larger, but eventually the performance deteriorates on *some*, in fact, infinitely many larger values of $\mathtt{n}$. For instance, consider an $\mathtt{n}$-digit number where we wish to see if it is a prime number or not. Clearly the task for an even $\mathtt{n}$ is easier than for $\mathtt{n}$ odd, in general. If a number is even and not equal to two, we immediately know it is not prime. All we have to do is check its least significant digit. However, for odd numbers we may have to work as much $\mathcal{O}_*\big(\sqrt{(}\mathtt{n})\big)$ operations. The worst-case takes place for prime numbers[1].

[1] We are going to study algorithms involving prime numbers, especially their significance in cryptography, soon.

Consider $g(n) = n^2(\sin(n) + 1)$ and $f(n) = n$. Which of the following relations are correct? Justify your answer.

A. $f(n) = \mathcal{O}_*\big(g(n)\big)$
B. $g(n) = \mathcal{O}_*\big(f(n)\big)$
C. Both A and B are correct.
D. Neither A or B is correct.

(Note I omitted to state that the definition is for $n$ *integer*. Otherwise, for instance, $|\log(x)| > x$ for infinitely many values between zero and one.)
C is correct. The function $\sin(n) + 1$ oscillates between 0 and 2. So $n^2(\sin(n) + 1)$ oscillates between $2n^2$ and zero. So for infinitely many integers $f(n) > g(n)$, and for infinitely many others $g(n) > f(n)$.

2. **The danger of using $\mathcal{O}(\cdot)$ notation in the exponent:** Suppose $f(n) = \mathcal{O}\big(g(n)\big)$. Is $2^{f(n)} = \mathcal{O}\big(2^{g(n)}\big)$? What if we replace $\mathcal{O}(\cdot)$ with $\Theta(\cdot)$? Justify your answer by either proving the claims, or providing a counterexample.

No. Consider $f(n) = n$ and $g(n) = 2n$. Then $f(n) = \mathcal{O}\big(g(n)\big)$ and $g(n) = \mathcal{O}\big(f(n)\big)$. Therefore, $f(n) = \Theta\big(g(n)\big)$. However,

$$2^{f(n)} = \mathcal{O}\big(2^{g(n)}\big) \quad \text{but} \quad 2^{g(n)} = 4^n \neq \mathcal{O}\big(2^{f(n)}\big)$$

In general, if $f(n) = \mathcal{O}\big(g(n)\big)$, but $g(n) \neq \mathcal{O}\big(f(n)\big)$, then $(1+\epsilon)^{f(n)} \prec (1+\epsilon)^{g(n)}$, for any constant $\epsilon > 0$. But if $f(n) = \Theta\big(g(n)\big)$, it does not imply that $2^{f(n)} = \Theta\Big(2^{g(n)}\Big)$

3. 3a) For two functions $f(\cdot)$ and $g(\cdot)$, we write $f \prec g$ if $f = \mathcal{O}(g)$ but $g \neq \mathcal{O}(f)$. And we write $f \approx g$ if $f = \mathcal{O}(g)$ and $g = \mathcal{O}(f)$, that is, if $g = \Theta(f)$. Write the following functions in increasing order from the "smallest" with respect to $\prec$. For instance, for $n, n^2, n^2 + n$ and $2^n$ we would write: $n \prec n^2 \approx n^2 + n \prec 2^n$. All $\log(\cdot)$ are in base 2, unless the base is specifically indicated. Also $\ln(\cdot)$ is the natural logarithm (base $e = 2.718182....$)

$$\log n, \frac{n}{2n+1}, \log_{10} n + \tfrac{\log n}{n}, \sin(n), n^3, (0.9)^n, 3^n, 2^{3\ln n}$$
$$n^{\log n}, n!, n^n, 1.000001^n, 3^{\sqrt{n}}, n^{1/3}, 1, \lfloor n \rfloor^2, \lceil n^2 \rceil$$

$$(0.9)^n \prec 1 \approx \frac{n}{2n+1} \approx c + \sin(n) \prec \log(n) \approx \log_{10}(n) \prec n + \frac{\log(n)}{n} \prec n\log(n) \prec n^{1/3} \prec$$
$$\lfloor n^2 \rfloor \approx \lceil n^2 \rceil \prec 2^{3\ln(n)} \prec n^3 \prec n^{\log(n)} \prec 3^{\sqrt{n}} \prec 1.000001^n \prec n! \prec n^n$$

**Comments:**

- The function $(0.9)^n$ is a decreasing functions and as $n \to \infty$, it goes to zero. Thus it is bounded from above by 1. However, 1 is constant and does not go to zero. No matter how big of a constant we choose for $C$, $C(0.9)^n$ will eventually fall below 1. Note that for any constant $c > 0$ we have $c = \Theta(1)$. However, $0 = \mathcal{O}(1)$ but $1 \neq \mathcal{O}(0)$.
- $\sin(n)$ is not comparable with 1, or $\frac{n}{2n+1}$ or $(0.9)^n$, since as $n \to \infty$, each one of these functions falls above, and falls below $\sin(n)$ infinitely many times. However, if you replace $\sin(n)$ by, $c+\sin(n)$ for some positive constant $c$, then $c+\sin(n) = \Theta(1)$. Also $\sin(n) \prec \log(n)$ and, in fact, for all functions $f(n)$ that are growing with $n$, no matter how slowly, we have $\sin(n) \prec f(n)$.
- $2^{3\ln(n)} = 8^{\ln(n)} = e^{\ln(8)\ln(n)} = n^{\ln(8)} = n^{2.079442\ldots}$.
- $n^{\log(n)} \prec (1+\epsilon)^{\sqrt{n}}$ for any positive constant $c$. To see this take the logarithms of both, and you will see that $\log\left(n^{\log(n)}\right) = \log^2(n)$, and $\log\left((1+\epsilon)^{\sqrt{n}}\right) = \log(1+\epsilon)\sqrt{n}$. A similar argument shows that $(1+\epsilon)^{\sqrt{n}} \prec (1+\delta)^n$ for any constants $\epsilon, \delta > 0$.

3b) Show that for two constants $a, b$ both strictly larger than 1, $\log_a(n) = \Theta\left(\log_b(n)\right)$.

We have

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$$

Since $\log_b(a)$ is a constant the claim is proved.

4. Your task is to express the Horner's algorithm for evaluating a polynomial $p(x) = p_0 + p_1 x + \cdots + p_n x^n$ at a given number $a$. So we wish to evaluate $p(a)$. However, you are to rewrite Horner's method **recursively**. More precisely, given a list (array) of $n+1$ $[p_0, p_1, \ldots, p_n]$ numbers and another number $a$ as input, return the number $p(a)$.

4a) Express your algorithm *recursively*. No `for, while` etc. loops are allowed. Only recursive calls and `if, if ... else` statements, and assignments may be used.

Suppose the polynomial $p_0 + p1_x + \cdots + p_n x^n$ is represented by its array of coefficients $[p_0, p_1, \ldots, p_n]$. The recursive Horner's algorithm could be:

**Algorithm**: Horner($[p_0, p_1, \ldots, p_n], a$)
**if** $n == 0$ **return** $p_0$
**else return** $p_0 + a * \text{Horner}([p_1, \ldots, p_n], x)$
**end**

4b) Let the worst-case number of arithmetic operations in your algorithm be $f(n)$ where $n$ is the number of items in the input. Find a recurrence relation for $f(n)$. Also state what $f(1)$ equals to.

The amount of work in the **else** part is $f(n-1)+c$. For $n = 0$, that is the **if** part, we have $f(0) = C$. So $f(n)$ is given by the recurrence: $f(0) = C$ and $f(n) = f(n-1) + c$.

4c) Solve the recurrence you derived and find the asymptotic running time of your recursive Horner's algorithm using both $\mathcal{O}(\cdot)$ and $\Theta(\cdot)$.

This is a simple tail recursion. We get $f(n) = \Theta(n)$.

5. Consider the recursive formulation of the *Binary Search* algorithm. Given a sorted and non-decreasing list of comparable objects L, and a new item x, find the location (index) of x or indicated that x is not in L.

5a) Give a recursive algorithm for binary search. No, **while, for, repeat** etc statements are allowed. Only **if, if else** and assignment statements are allowed.

The recursive characterization of binary search is:

**Algorithm**: *Bsearch*($[L_m, L_{m+1}, \ldots, L_n], k$)
1) **If** n==m **and** $L_m == k$ **return** (m,$L_m$)
2) **If** n==m **and** $L_m != k$ **return** *Not Found!*
3) **If** $L_{\lfloor \frac{n+m}{2} \rfloor} == k$ **return** $\left(\lfloor \frac{n+m}{2} \rfloor, L_{\lfloor \frac{n+m}{2} \rfloor}\right)$
4) **If** $L_{\lfloor \frac{n+m}{2} \rfloor} < k$ **return** *Bsearch* $\left([L_{\lfloor \frac{n+m}{2} \rfloor + 1}, \ldots, L_n], k\right)$
5) **If** $L_{\lfloor \frac{n+m}{2} \rfloor} > k$ **return** *Bsearch* $\left([L_m, \ldots, L_{\lfloor \frac{n+m}{2} \rfloor}], k\right)$

5b) Write a difference equation for the running time of your *Binary Search* algorithm. Solve your equation and express the performance of the algorithm in $\Theta(\cdot)$ notation.

To run the binary search on a list $[L_1, L_2, \ldots, L_n]$ and a key $k$ the call is *Bsearch*($[L_1, \ldots, L_n], k$), so $m = 1$. Since there are $n$ items in the ordered list we need to find the recurrence relation $f(n)$ for the

number of comparisons and other overhead. In steps 1), 2) and 3) the amount of operations is constant. In steps 4) and 5) we call *Bsearch* on list of at most half the size, so the amount of work is $f(n/2)$. **Note**, however, that at each recursive call only one of steps 4) or 5) is run. So the recurrence is given by:

$$f(n) = f\left(\tfrac{n}{2}\right) + c$$

Using the master theorem, we get $f(n) = \mathcal{O}\big(\log(n)\big)$. With direct computation write $n = 2^k$ and $g(k) = f(2^k)$. Then the recurrence turns into the tail recursion: $g(k) = g(k-1) + c$. This is easily seen to be $g(k) = ck$. Therefore, $f(n) = f(2^k) = ck = c\log(n)$.