COMP 250 2016, Assignment 2
Due Thursday February 18th 2016

(no programming this time)

[6%] 1. Prove that if

$$\lim_{n\to\infty} f(n)/g(n) = 0$$

then f(n) **is** O(g(n)) but f(n) **is not** θ(g(n)).

> By definition, "$\lim_{n\to\infty} f(n)/g(n) = 0$" means that for all $\epsilon>0$ there exists n_0 s.t. $n>n_0$ implies $f(n)/g(n)<\epsilon$. Choose $c=\epsilon=1$ and let n_0 be given by the limit definition. Then $n>n_0$ implies $f(n)<1\bullet g(n)$. This means f(n) is O(g(n)).
>
> Now we show f(n) is not Ω(g(n)). We wish to prove that for all $c>0$ and for all $n_0>0$ there exists $n>n_0$ s.t. $f(n)<cg(n)$. But this is weaker than what we get from the limit definition: for all $c=\epsilon>0$ there exists n_0 s.t. $n>n_0$ implies $f(n)<cg(n)$.

2. **Solve the following Exercises**

[5%] Suppose you have algorithms with the five running times listed below. (Assume these are the exact running times.) How much slower do each of these algorithms get when you (a) double the input size, or (b) increase the input size by one?

a) 99n
b) n^2
c) n^4
d) $n2^n$
e) 3^n

a) $99(2n)/99n = \mathbf{2}$	$99(n+1)/99n = \mathbf{1} + {}^{1}/_{n}$
b) $(2n)^2/n^2 = \mathbf{4}$	$(n+1)^2/n^2 \approx \mathbf{1} + {}^{2}/_{n}$
c) $(2n)^4/n^4 = \mathbf{16}$	$(n+1)^4/n^4 \approx \mathbf{1} + {}^{4}/_{n}$
d) $(2n)2^{2n}/n2^n = \mathbf{2^{n+1}}$	$(n+1)2^{n+1}/n2^n = \mathbf{2} + {}^{2}/_{n}$
e) $3^{2n}/3^n = \mathbf{3^n}$	$3^{n+1}/3^n = \mathbf{3}$

[5%]

Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

f) $f_1(n) = 99n^2$
g) $f_2(n) = 2^{\lg n}$
h) $f_3(n) = n^2 \log\log n$
i) $f_4(n) = n2^n$
j) $f_5(n) = 3^n$

$O(2^{\lg n})=O(n)\subsetneq O(99n^2)\subsetneq O(n^2 \log\log n)\subsetneq O(n2^n)\subsetneq O(3^n)$

$\lim_{n\to\infty} n/99n^2 = \lim_{n\to\infty} 1/99n = 0$
$\lim_{n\to\infty} n^2 / n^2 \log\log n = \lim_{n\to\infty} 1/\log\log n = 0$
$\lim_{n\to\infty} n^2 \log\log n/ n2^n = \lim_{n\to\infty} n \log\log n/2^n = 0$
$\lim_{n\to\infty} n2^n/3^n = \lim_{n\to\infty} n/(3/2)^n = 0$

[6%]

3. Assume you have functions f and g such that $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

(a) $\log_2 f(n)$ is $O(\log_2 g(n))$.
(b) $2^{f(n)}$ is $O(2^{g(n)})$.
(c) $f(n)^2$ is $O(g(n)^2)$.

(a) False. Take $f(n) = 2$ and $g(n) = 1$.
(b) False. Take $f(n) = n$ and $g(n) = n/2$. Then $\lim_{n\to\infty} 2^{g(n)}/2^{f(n)}=\lim_{n\to\infty} 2^{n/2}/2^n = \lim_{n\to\infty} 1/2^{n/2} = 0$.
(c) True. $f(n)<cg(n)$ implies $f(n)^2<c^2g(n)^2$.

4.

[8%]

For each of the algorithms below, indicate the running time using the simplest and most accurate big-Oh notation. Assume that all arithmetic operations can be done in constant time. The first algorithm is an example. No justifications are needed.

Algorithm	**Running time in big-Oh notation**
Algorithm Example(n) $x \leftarrow 0$ **for** $i \leftarrow 1$ **to** n **do** . $x \leftarrow x + 1$	**O(n)**
Algorithm algo1(n) $i \leftarrow 1$ **while** $i < n$ **do** . $i \leftarrow i + 100$	$O(n)$
Algorithm algo2(n) $x \leftarrow 0$ **for** $i \leftarrow 1$ **to** n **do** . **for** $j \leftarrow 1$ **to** i **do** . $x \leftarrow x + 1$	$O(n^2)$
Algorithm algo3(n) $i \leftarrow n$ **while** $(i > 1)$ **do** . $i \leftarrow i/2$	$O(\log_2(n))$
Algorithm algo4(n) $k \leftarrow 1$ **for** $i \leftarrow 1$ **to** 1000 . **for** $j \leftarrow 1$ to i . $k \leftarrow (k + i - j) * (2 + i + j)$	$O(1)$

[10%]

5. We know $\sum_{i=1}^{n} i = n(n+1)/2$, $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$, $\sum_{i=1}^{n} i^3 = (n(n+1)/2)^2$. Prove that in general $\sum_{i=1}^{n} i^k$ **is** $\Theta(n^{k+1})$ for any positive integer k.

Proof. $\sum_{i=1}^{n} i^k$ **is** $O(n^{k+1})$: $\sum_{i=1}^{n} i^k < \sum_{i=1}^{n} n^k = n^{k+1}$

$\sum_{i=1}^{n} i^k$ **is** $\Omega(n^{k+1})$: $\sum_{i=1}^{n} i^k > \sum_{i=\lfloor n/2 \rfloor}^{n} i^k$

$> \sum_{i=\lfloor n/2 \rfloor}^{n} (n/2)^k$

$\geq (n/2)^{k+1} = cn^{k+1}$

for $c=1/2^{k+1}$.

[5%] 6.

Find two non-negative functions f and g such that $f(n)$ is not $O(g(n))$ and $g(n)$ is not $O(f(n))$.

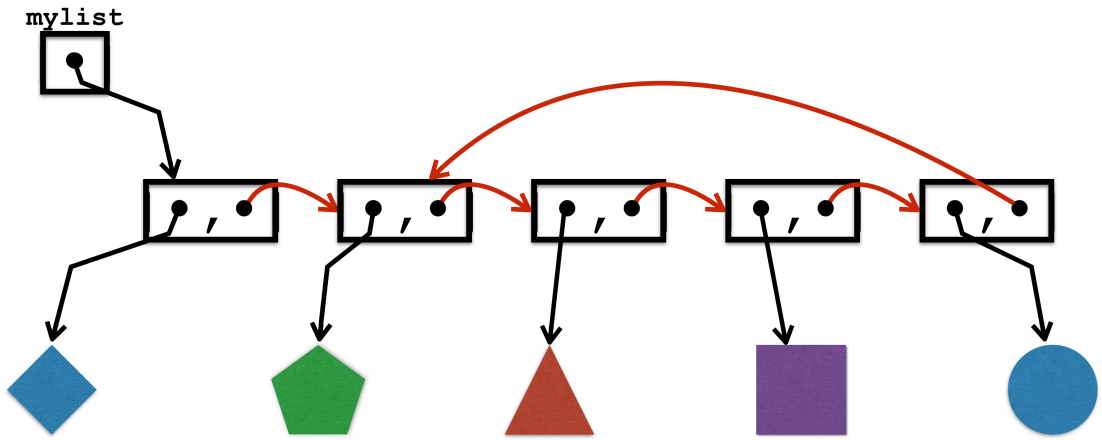
Proof. Define

$$f(n)=\begin{cases}1 & \text{if n is even}\\ n & \text{if n is odd}\end{cases}$$

$$g(n)=\begin{cases}n & \text{if n is even}\\ 1 & \text{if n is odd}\end{cases}$$

Clearly f(n) is not O(g(n)) for odd n, g(n) is not O(f(n)) for even n.

7. Suppose I give you access to a (singly) linked list by its entry point named "**`mylist`**". You can move from one cell to the next by using the "**`.next`**" field of each cell. For example, **`mylist`** points at the cell containing the blue diamond, whereas **`mylist.next`** points at the cell containing the green pentagone. If you walk through this list from the head forward you will notice that it contains no null pointer. This means that at some point this list points into itself. In the example below, you can see this because you see the entire list from the outside. But if you visit all the nodes of the list, one by one, it maybe a bit more difficult to figure out what is going on.



Now imagine you start two pointers at the beginning of the list: one named "**tortoise**" and another named "**hare**". The former moves step by step while the latter moves two steps at a time

```
tortoise = tortoise.next
hare = hare.next.next
```

Repeat this process until the two pointers meet : **tortoise == hare**.

[3%] a) Where will this occur in the example above ?

> At the blue-circle. (sorry for the mistake)

Once this has happened, move the **tortoise** back to the beginning of the list. Start moving the two pointers from their current positions in the list, but both in a step by step fashion:

```
tortoise = tortoise.next
hare = hare.next
```

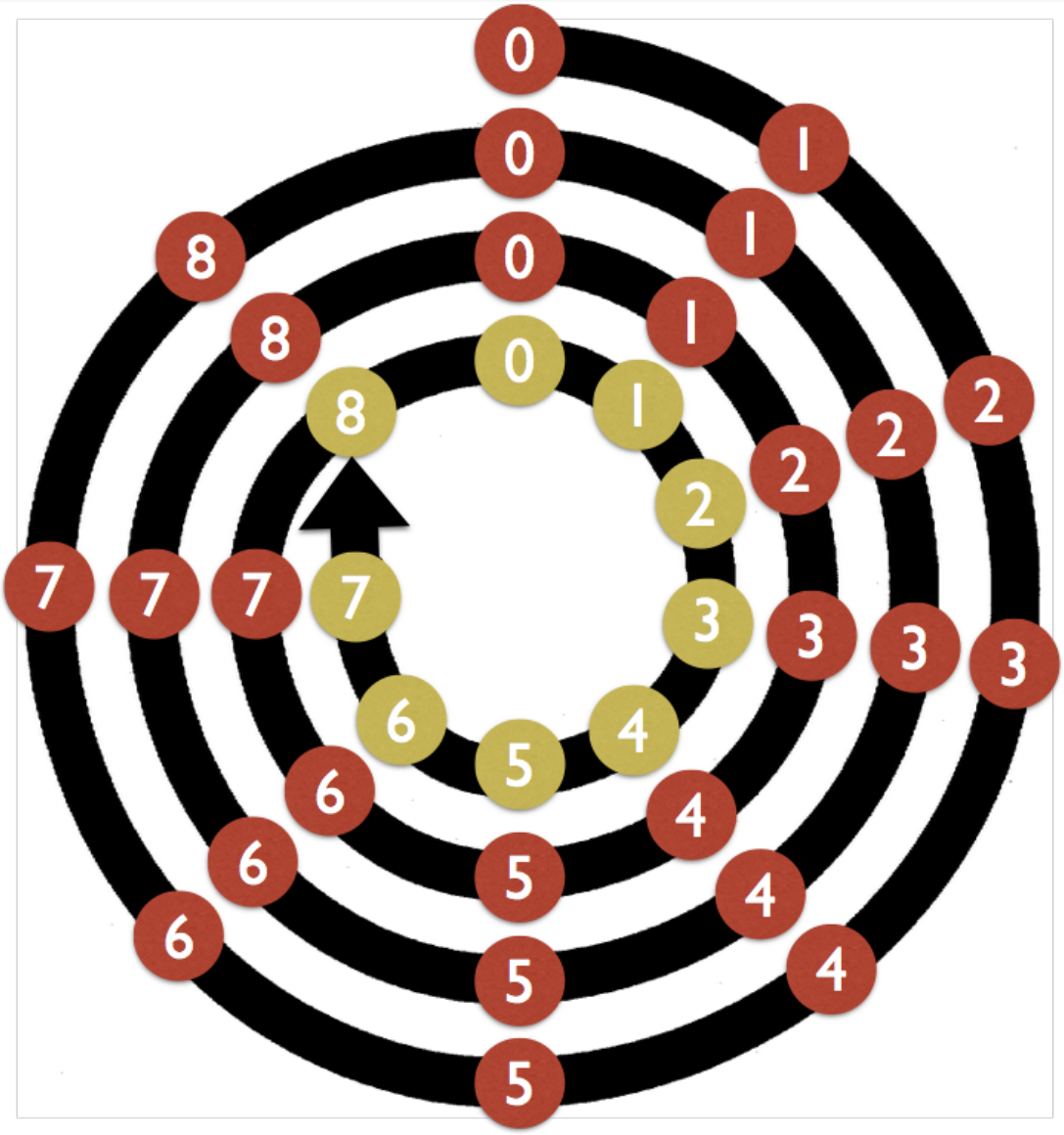
Repeat until the two pointers meet once again : **tortoise == hare**.

[3%] b) Where will this occur in the example above ?

> At the green-pentagone.

[8%] c) Show that whatever is the length **x** of the initial segment before entering the loop and whatever is the length **y** of the loop itself, the above process will always find the point connecting the two parts. Provide a formal proof here.

> The red dots form the tail and the yellow dots form the loop. Let Y be the length of the loop. Pretend we know Y and write down the numbers %Y on the dots starting with zero at the beginning of the tail. You get the picture below.

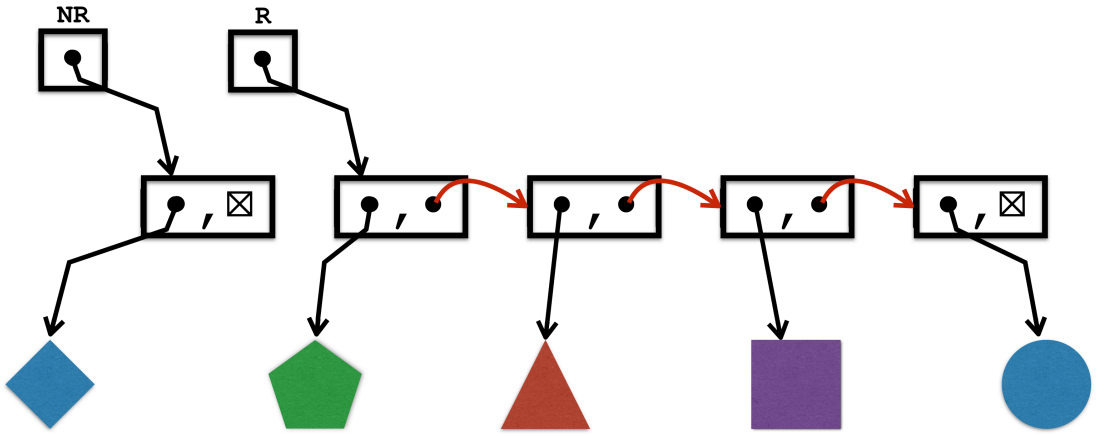


After K steps of the algorithm, the tortoise pointer will be on a node labeled with K%Y and the hare on 2K%Y. The two pointers cannot meet unless K is large enough to reach the yellow part. Once K is large enough to reach the yellow nodes, the pointers meet exactly when 2K%Y = K%Y = 0. That's because the simpler condition 2K%Y = K%Y implies K%Y = 0 (by subtracting K from both sides).

Thus when we return the tortoise at the initial zero, the hare pointer is on the innermost zero. Now both pointers move one step at a time, meaning that they stay synchronized %Y as they move. The tortoise circling down the spiral, while the hare circles the loop at the core of this structure. As soon as the tortoise pointer reaches the loop (yellow 8 in this example), the hare pointer will join in because they are always synchronized %Y. Thus they meet at the place we wanted them to meet: the unique dot where tail and loop join together. All that mattered is that they were both on a zero %Y at the end of the first phase of the algorithm, and that they stayed synchronized %Y throughout the second phase of the algorithm.

[10%] d) Combine all this together into an algorithm that breaks an arbitrary list **mylist** into two sub-lists:

NR = the part of the list from beginning before the connecting point
R = the part of the list closing into a loop starting at the connecting point.



Make sure your algorithm works even if the list does not contain a loop, in which case **NR** contains the whole list and **R** is an empty list.

```
NR = mylist;
IF NR!=NULL THEN { tortoise = NR.next;
    IF NR.next!=NULL THEN hare = NR.next.next}
WHILE
(hare!=NULL && hare.next!=NULL && hare!=tortoise)
DO {tortoise = tortoise.next
    hare = hare.next.next}
IF hare==NULL OR hare.next==NULL THEN R=NULL
ELSE{tortoise = NR
    WHILE (hare!=tortoise) DO {
        tortoise = tortoise.next
        hare = hare.next}
    IF NR==hare THEN {NR=NULL}
    ELSE{WHILE mylist.next!=hare DO
            mylist=mylist.next
        mylist.next=NULL
        }
    WHILE tortoise.next!=hare DO
        {tortoise=tortoise.next}
    tortoise.next=NULL
    R=hare
    }
```

[8%] e) Analyse the running time of your algorithm as a function of the number of cells in the list. How much space does your algorithm use as a function of the number of cells in the list.

Let x be the length of the tail and y the length of the loop.
Total time **is** $O(x+y)$ and space is $O(1)$.

[10%] f) Let **f(s)** be a function from a finite set **S** into itself. Define for all **s**∈**S**, $\mathbf{f^{(0)}(s) = s}$ and $\mathbf{f^{(n)}(s) = f(f^{(n-1)}(s))}$ for **n>0**. Give an efficient algorithm to identify for each **x** the smallest **n** and **k** such that $\mathbf{f^{(n)}(x) = f^{(n+k)}(x)}$.

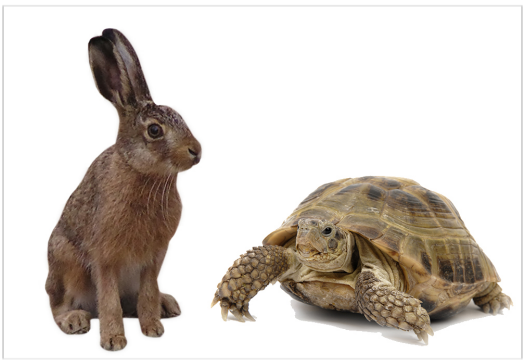
```
tortoise = f(x); hare = f(f(x));
WHILE (hare!=tortoise) DO
     {tortoise = f(tortoise);
     hare = f(f(hare))}
tortoise = x
WHILE (hare!=tortoise) DO {
     tortoise = f(tortoise)
     hare = f(hare)}
n=0
IF x!=hare THEN   {
                  n=1
                  WHILE f(x)!=hare DO
                       {x=f(x); n=n+1}
                  }
k=1
WHILE f(tortoise)!=hare DO
     {tortoise=f(tortoise); k=k+1}
```

[8%] g) Analyse the running time of your algorithm as a function of **n** and **k**. How much space does your algorithm use as a function of **n** and **k**.

Let n be the length of the tail and k the length of the loop.
Total time **is** $O(n+k)$ and space is $O(1)$.

[5%] h) Explain the link between this whole problem and what was asked in HW1 !!!

In HW-1, when computing the periodic part of the fractional representation we most likely used an array to store all the remainders encountered so far. Most likely this lead to an $O(n)$ space and $O((n+k)^2)$ time solution. Following the guidelines of the current exercise, Q7f) in particular, an $O(1)$ space and $O(n+k)$ time solution could have been constructed. In HW-1 the function f was defined by computing the next remainder of the division.



Le Lièvre et la Tortue

Rien ne sert de courir ; il faut partir à point.
Le Lièvre et la Tortue en sont un témoignage.
Gageons, dit celle-ci, que vous n'atteindrez point
Sitôt que moi ce but. - Sitôt ? Êtes-vous sage ?
Repartit l'animal léger.
Ma commère, il vous faut purger
Avec quatre grains d'ellébore.
- Sage ou non, je parie encore.
Ainsi fut fait : et de tous deux
On mit près du but les enjeux :
Savoir quoi, ce n'est pas l'affaire,
Ni de quel juge l'on convint.
Notre Lièvre n'avait que quatre pas à faire ;
J'entends de ceux qu'il fait lorsque prêt d'être atteint
Il s'éloigne des chiens, les renvoie aux Calendes,
Et leur fait arpenter les landes.
Ayant, dis-je, du temps de reste pour brouter,
Pour dormir, et pour écouter
D'où vient le vent, il laisse la Tortue
Aller son train de Sénateur.
Elle part, elle s'évertue ;
Elle se hâte avec lenteur.
Lui cependant méprise une telle victoire,
Tient la gageure à peu de gloire,
Croit qu'il y va de son honneur
De partir tard. Il broute, il se repose,
Il s'amuse à toute autre chose
Qu'à la gageure. A la fin quand il vit
Que l'autre touchait presque au bout de la carrière,
Il partit comme un trait ; mais les élans qu'il fit
Furent vains : la Tortue arriva la première.
Eh bien ! lui cria-t-elle, avais-je pas raison ?
De quoi vous sert votre vitesse ?
Moi, l'emporter ! et que serait-ce
Si vous portiez une maison?