CS2800 Design and Analysis of Algorithms
Problem Set #0 January 24, 2023

1. Recall the definition of Big-Oh: We say that $f(n) = O(g(n))$ if $\exists c > 0, n_0 > 0$ such that for every $n \geq n_0$, $f(n) \leq cg(n)$. Show that the definition given above is equivalent to the following definition.

   $f(n) = O(g(n))$ if $\exists c > 0$ such that for every $n \geq 1$, $f(n) \leq cg(n)$.

   Solution: Consider the set of values $\{\frac{f(1)}{g(1)}, \frac{f(2)}{g(2)}, \ldots, \frac{f(n_0)}{g(n_0)}\}$. If I choose the largest of this as a constant c', then $f(n) \leq c'g(n)$ for all $n \leq n_0$. This should help you prove the equivalence.

2. For the following functions $f(n)$ and $g(n)$, answer what is the tightest asymptotic relation between $f(n)$ and $g(n)$.
   (a) $f(n) = n^{\log_2 c}$ and $g(n) = c^{\log_2 n}$ where $c > 2$ is a constant.
   (b) $f(n) = 2^{n^2}$ and $g(n) = n^2 2^{2n}$.
   (c) $f(n) = (\log n)!$ and $g(n) = n^2$.
   (d) $f(n) = n^{\frac{\log\log n}{\log n}}$ and $g(n) = n^{1/10}$.

   Solution: Use the limit definitions used in class for all these problems, as well as some standard algebraic manipulations.

3. Is $2^{O(n)} = O(2^n)$? If yes, give a proof. Otherwise, give a counter-example.

   Solution: Consider the functions 4^n and 2^n.

4. Is it true that for all functions $f(n)$, $f(n) = \Theta(f(n/2))$? If yes, give a proof. Otherwise, give a counter-example.

   Solution: Consider the example from the previous problem.

5. Does there exist an $\epsilon > 0$ such that $\log n = \Omega(n^\epsilon)$?

Solution: Q2, Part(d) is the case for $\epsilon = 1/10$.

6. You have devised a new algorithm that you title *ternary search*. The algorithm is as follows:

```
TernarySearch(A, k)
  Input: Sorted array A[1, 2, ..., n] and a key k
  Output: Index of k, if present, else −1
1 if A is empty then return −1
2 if A[n/3] = k then
3     return n/3
4 else
5     if A[2n/3] = k then
6         return 2n/3
7     else
8         if k < A[n/3] then
9             Ternary Search(A[1, 2, ..., n/3 − 1])
10        else
11            if A[n/3] < k < A[2n/3] then
12                Ternary Search(A[n/3 + 1, ..., 2n/3 − 1])
13            else
14                Ternary Search(A[2n/3 + 1, ..., n])
```

Is your algorithm better than binary search? Why/Why not. Support your reasons mathematically.

Solution: The recurrence for the worst-case running time for this algorithm would be

$$T(n) = T\left(\frac{n}{3}\right) + \Theta(1)$$

7. Recall the *Tower of Hanoi* problem: You are given n disks on a peg (peg 0), and two additional pegs (peg 1 and 2). You goal is to move all the disks from peg 0 to peg 1 or 2 under the following constraint: You can only move one disk at a time, and you are not allowed to place a larger disk above a smaller disk.

   (a) Describe an algorithm to perform the task (this will be recursive). Write a recurrence relation for the number of movements of disks your algorithm makes, and compute the value exactly.

Solution: If $T(n)$ is number of steps to move n disks from one peg to another satisfying the constraints, then we can write

$$T(n) = 2T(n-1) + 1$$

First $T(n-1)$ to move the top $n-1$ disks to another peg, then 1 step to move the final disk to the remaining peg, and then $T(n-1)$ moves to move all the $n-1$ disks on top of the final disks.

Now unroll the recurrence and guess that $T(n) = 2^n - 1$, and verify the same using induction.

(b) Consider the following additional constraint to the problem: You are only allowed to move a disk from peg 0 to 1, 1 to 2 or 2 to 0. Design an algorithm to move all the disks from peg 0 to one of the other pegs with this additional constraint. How many movements of disks does your algorithm make now?

Solution: This problem turned out to be much harder than I thought. It is a little out of the scope of the material covered presently. I am just outlining the recurrence and one way to proceed with it.

Let $t_1(n)$ be the number of steps needed to move n disks from peg i to $i+1 \pmod 3$ and let $t_2(n)$ be the number of steps needed to move n disks from i to $i+2 \pmod 3$. Now, we can write the following recurrences.

$$t_1(n) = 2t_2(n-1) + 1, \text{ and}$$
$$t_2(n) = 2t_2(n-1) + t_1(n-1) + 2,$$

with $t_1(1) = 1$, $t_2(1) = 2$ etc. Now, you could rewrite the recurrence for $t_2(n)$ as

$$t_2(n) = 2(t_2(n-1) + t_2(n-2)) + 3$$

You need to use characteristic equations or generating functions to solve this exactly. If you know this from your CS1200 material, you can try using that.

8. Smullyan[1] Island has three types of inhabitants: *knights*, who always speak the truth, *knaves* who always lies, and *normals* who speak truth sometimes, and lie sometimes. Everyone knows everyone else's names and type. Your goal is to know everyone's type. You can ask any inhabitant, of another inhabitant's type (this is the only question you are allowed to ask), but you cannot ask his/her own type. Asking the same question multiple times to a person will yield the same answer.

[1]Raymond Smullyan was a mathematician and philosopher famous for his many logic puzzles using self-reference involving knights and knaves. Check out What is the name of this book?

(a) Suppose that a strict majority of inhabitants are knights, give an efficient algorithm to find the type of every inhabitant.

> **Solution:** For a fixed person x, ask everyone else about his/her type. The answer given by at least half the number of people must be correct. If you find a knight, ask that person for the type of every other person.
>
> You could think about the most efficient ways to do this (in terms of the number of questions asked). I don't have a clear answer for it. If you find an interesting solution, then let me know.

(b) Prove that if the number of knights is at most half the total number of inhabitants, then it is impossible to find the type of every inhabitant correctly.

> **Solution:** Consider the situation where there are $2n$ inhabitants of which there are n knights and n knaves. Assume that all the knaves say that the knights are normals, and say that they themselves are knights.
>
> Consider another situation where there are $2n$ inhabitants of where there are n knights and n normals. Assume that all the normals say that the knights are knaves, and say that they themselves are knights.
>
> Consider any algorithm that tries to find the type of each inhabitant. The input for this algorithm will be the same in both the scenarios. How will the algorithm distinguish the cases?