CS 330 – Fall 2018 – Assignment 1 Solutions

Problem 1. (10 pts) Chapter 1, problem 8.

Solution 1. In case we run the man (hospital) -optimal version of the Gale-Shapley algorithm, there is no incentive for the men to state false preferences. We did this proof in class (slide 19 on stable matchings). On the other hand, there are examples where a woman can improve her match by misrepresenting her preferences. See the below tables that show the preferences of the men and women for such an example.

preference list of men:

	1^{st}	2^{nd}	3^{rd}
A	X	Y	Z
B	Y	X	Z
C	X	Y	Z

preference list of women:

	1^{st}	2^{nd}	3^{rd}
X	B	A	C
Y	A	B	C
Z	A	B	C

X lies:

	1^{st}	2^{nd}	3^{rd}
X	B	C	A
Y	A	B	C
Z	A	B	C

If we run Gale-Sahpley on the true preference list, then we get the assignments X–A, Y–B, Z–C. Here X is assigned to her second preference. However, if X lies and says she prefers C over A (but she actually doesn't), then she'll end up with her first preference, that is X–B, Y–A, Z–C.

Problem 2. (10 pts) In this problem we ask you to refresh your knowledge on asymptotic notations.

Rank the following functions by order of growth; that is, find an arrangement of the expressions in increasing order of there $O(\ )$ (big-Oh) values. (Note: below, $\lg n$ means $\log_2 n$). The answer you need to submit for this problem is simply an ordered list of the expressions below.

n^{100}	$(\sqrt{2})^{\lg n}$	n^2	$\sqrt{n}$	$2^{2^{n+1}}$
n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\lg n$	$2^{100 \lg n}$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\log_3 5n$
$\binom{n}{2} + n \lg n$	$4^{\lg n}$	n	$n \lg n$	$2^{\lg^{1.001} n}$

Solution 2. The list of functions in non-increasing asymptotic order is:

1. $2^{2^{n+1}} = 2^{2 \cdot 2^n} = 4^{2^n}$
2. 2^{2^n}
3. $n2^n = 2^{\lg n} \cdot 2^n = 2^{n + \lg n}$
4. $2^{\lg^{1.001} n} = 2^{(\lg n) \lg^{0.001} n} = n^{\lg^{0.001} n}$

5. $n^{\lg\lg n} = 2^{(\lg n)\lg\lg n}$
6. n^{100}
7. $2^{100\lg n} = 2^{\lg n^{100}} = n^{100}$
8. n^3
9. n^2
10. $4^{\lg n} = 2^{2\lg n} = 2^{\lg n^2} = n^2$
11. $\binom{n}{2} + n\lg n = n(n-1)/2 + n\lg n = \Theta(n^2)$
12. $n\lg n$
13. $\lg(n!) = \Theta(n\lg n)$ (this can be done by observing that $\lg n!$ = $\lg 1 + \lg 2 + \cdots + \lg n$, so $n/2\lg(n/2) \leq \lg(n!) \leq n\lg n$)
14. n
15. $\sqrt{n}$
16. $\left(\sqrt{2}\right)^{\lg n} = 2^{\frac{1}{2}\lg n} = \sqrt{n}$
17. $\lg^2 n$
18. $\lg n$
19. $\log_3 5n = \log_3 5 + \log_3 n = \Theta(\lg n)$
20. $n^{1/\lg n} = 2^{\frac{\lg n}{\lg n}} = 2$

From the list we observe the equivalence classes:

1. n^{100} and $2^{100\lg n}$
2. $n^2, 4^{\lg n}$, and $\binom{n}{2} + n\lg n$
3. $n\lg n$ and $\lg(n!)$
4. $\sqrt{n}$ and $\left(\sqrt{2}\right)^{\lg n}$
5. $\lg n$ and $\log_3 5n$

The correctness of the list is easily observed by taking the ratio of each function with the next function and checking the limit as n goes to infinity. For instance, ratio of the first function to the second is 2^{2n}, which goes to infinity as n goes to infinity. Most ratios are easy to resolve through basic algebra and observation such as $lg^{\alpha}n = o(n^{\beta})$ for any positive constants α, β (this can be shown by L'Hôpital's rule).

It is not easy to place $n^{\lg\lg n}$. Observe that the exponent here is $\lg\lg n$, so it eventually it outgrows any constant, like 100: $n^{\lg\lg n}/n^{100} = n^{\lg\lg n - 100}$. As n grows, so does $\lg\lg n - 100$, so the whole thing goes to ∞.

Some people have problems placing $2^{\lg^{1.001} n}$. Again, it is equal to $2^{(\lg n)\lg^{.001} n} = n^{\lg^{.001} n}$. And again, the exponent $\lg^{.001} n$, although it grows slowly, eventually outgrows any constant. It even outgrows $\lg\lg n$: letting $k = \lg n$, we get that $\lg^{.001} n = k^{.001}$, while $\lg\lg n = \lg k$, and $\lg k$ grows slower than k to any power, even if the power is as small as .001.

Problem 3. (10 pts) Chapter 3 problem 5 on page 108.

Solution 3. We proof this claim by *induction* on the *number of nodes* n in the graph. Let $T = (V, E)$ be a binary tree. We denote by L the number of leaves in the graph and by Z the number of nodes with exactly two children. Then we want to proof that $Z = L - 1$. Note, that we do not require T to be complete.

Base case: Let us first assume $n = 2$ and T consists of a single edge. In this "tree" we get $L = 1$ (consider one node as the root and the other as the leaf) and $Z = 0$, thus the claim holds.

Inductive assumption: Assume that for any binary tree T with $|V| = n - 1$ nodes it holds that $Z = L - 1$.

Proof for n: We now show that if the inductive assumption holds, then the formula is also true for any binary tree T of size $|V| = n$. Let v be a leaf node in V. Remove v from the graph to get $T^{-v} = (V - \{v\}, E - \{e(v)\})$. Here $e(v)$ represents the single edge that is adjacent to v in T. By the inductive assumption we know that in T^{-v} the formula is true. Consider the location of node v in T; let u be the parent of v in T (thus $e(v) = (u, v)$). Because T is binary, node u can have 0 or 1 child except for v.

case 1: if u has no other child, then it is a leaf in T^{-v}. Then in T, it is a node with one child, and instead v becomes a leaf. As a result neither Z, nor L is changed, thus the formula still holds.

case 2: node u has one other child. Then it does not contribute to the formula in T^{-v} on either side. Now, consider u in T. Here u has exactly two children, hence Z is increased by one. However, v is an additional leaf, that was not present in T^{-v}, thus the number of leaves L is also increased by one. Because both sides increase by one, the formula still holds. □

Problem 4. (20 pts) This is your first programming assignment in this semester. To see in what format we expect you to submit your solution, please see the **guidelines** posted on piazza.

Your program will answer the question whether a given sequence of integers contains three elements that sum to zero. The two files you create should be named `zerosumsort.py` (or java file) and `zerosumfaster.py` (or java file).

Your code should take as input a sequence of n integers $x_1, x_2, \ldots, x_n$ (each integer may be positive or negative and the sequence may contain duplicates), entered from the command line and separated by commas. The correct output is a set of three distinct indices $\{i, j, k\}$ such that $x_i + x_j + x_k = 0$ if such a set exists, and "No" otherwise.

a) Start by sorting the input, and devising an algorithm to run in O($n^2 \log n$) time. Please add the appropriate comments to your code to justify the running time.
Submit in `zerosumsort.py` (or java file)

b) It is possible to speed up your algorithm to run in $O(n^2)$ time by using a data structure from CS112.
Build such a solution, adding the appropriate comments in your code to justify the running time.
Submit in `zerosumfaster.py` (or java file)

Solution 4.
$O(n^3)$ **solution:** The brute-force approach to this problem is to test all triples (x_i, x_j, x_k) for their sum. The number of triples is $\binom{n}{3}$, hence this would yield a cubic algorithm.

$O(n^2 logn)$ **solution:** [implemented in `zerosumsort.py`] A major improvement in running time can be achieved by only iterating over all tuples (x_i, x_j). Since we want to find triples that sum to zero, we know that the only acceptable value for x_k would be $x_k = -(x_i + x_j)$. So, instead of trying each potential value for x_k we check whether $-(x_i + x_j)$ is in the list. If the list is in sorted order, then this can be done in $O(logn)$ time (with binary search for example). As a preprocessing step we sort the list in $O(n \log n)$ time. You can find this implementation in `zerosumsort.py` The total running time of this code is $O(n^2 logn)$ since the nested for loop takes $O(n^2)$ iterations and there is a $O(logn)$ search within the loop and the sorting in the beginning is only an additional factor.
$O(n^2)$ **solution:** [implemented in `zerosumsortfaster.py`] In the previous implementation we spent $O(n \cdot logn)$ time to sort the numbers and then again in each iteration we search for k in $O(logn)$ time. We can use a

hashtable (or dictionary) instead of the sorted list to speed things up. Remember, that inserting a value into it takes $O(logn)$ time, thus building the entire map of the sequence takes $O(n \cdot logn)$ time. However, the lookup after this takes only constant. In addition $O(n \cdot logn)$ is trumped by the $O(n^2)$ iterations of the for loops.