CS3230 : Design and Analysis of Algorithms (Fall 2014)

REMARKS ON HOMEWORKS
&
ADVICE ON HOW TO SOLVE THEM

(Leong Hon Wai, Revised: Aug 2014)

Quote: *Algorithms lie in the heart of computer science.* (Aho, Hopcroft, Ullman, 1974.)

About the Tutorial and Homework Problems in CS3230

Generally, the tutorial and homework problems are designed to assist you in understanding and applying the course materials. I provide three types of questions:

- **R-problems** (*Routine* problems),
- **S-problems** (*Standard* problems), and
- **A-problems** (*Advanced* problems).
-

You are required to answer all the *"Standard problems"* (S-problems) in each homework set and turn them in by the due-date. The S-problems give a rough indication of the benchmark standard of the course.

The *Routine problems* (**R-problems**) are provided as practice material and to aid you in understanding the basics of the lectures, definitions, algorithms, proofs, etc. These are for you to try out on your own; do not turn these in. If you cannot solve these problems, then it usually mean you are not following (understanding) the course. REVISE, and FIND HELP QUICKLY. If you consistently cannot solve the R-problems, quickly talk to the teaching staff for help.

In addition, there will be some *Advanced problems* (**A-problems**) that are more challenging problems. *The A-questions are completely optional; we do not expect many students to try them. No extra credit will be given to A-problems* (*as doing so will disadvantage the other students*). But Prof. Leong gives out "coffee awards" to solutions or good attempts to these A-problems.

Academic Policy

For all S-problems (to be submitted), you are supposed to work on the problems INDIVIDUALLY and submit YOUR OWN solutions.

Discussion among students MUST BE restricted to only discussion on general approaches; each student must write out their own individual solution to the problem and you must write down their names of the collaborators for that problem. There MUST be no comparison of solutions, or copying of solutions or of the detailed steps.

Comparing or copying solutions write-up or detailed steps is considered **cheating** or **plagiarism**. Not listing out the names of people you discuss with is also considered an attempt at plagiarism.

Plagiarism and other anti-intellectual behaviour will not be tolerated.
The University takes a strong stand on this and students are warned of dire consequences.

If you need help, you should have, at least have worked INDIVIDUALY on the problem for some time before you even venture to get help. You can approach the TA or the instructors for help during their announced office hours. Other times can be arranged via email -- however, advanced notice should be given. Responsibility starts with *you*.

Discussion on R-problems and S-problems after the submission deadline.

For R-problems (that you do not hand in), you are free to collaborate ad-hoc or in study groups. For the S-problems after the homework deadline, you are also encouraged to discuss different solutions to the problems, discuss their pros and cons. In general, there may be many different ways of solving a given problem. Also, some solutions may be more concise than others. Whenever a *simple* solution suffices, give a simple solution. Finally, the interested students may also want to explore variations or generalizations of the problem and/or the solutions.

About Your Solutions (IMPORTANT!)

Answers to homework should be written at a level suitable for and to address the instructor. In other words, don't put in long explanation for trivial things. Learn to make appropriate definitions to make your answers more precise and concise. If you want to organize the set S (of people) as a linked-list, just say so. If you need to sort an array Q, just say "sort array Q".

You will be graded not only on the correctness and efficiency of your answers, but also on the clarity of your solution. Therefore, *it pays to make your answers more concise and clearer.* Understandability of the solution is as desirable as correctness. Sloppy answers will be at a disadvantage, i.e. likely to receive fewer points, even if they are correct.

When asked to "give an algorithm" to solve a problem, your write-up should *NOT* be just the code. (*This will receive very low marks*.) Instead, it should be a short essay. The first paragraph should summarize the problem and what your results are. The body of the essay should consist of the following:

- A description of the algorithm *in English* and, if helpful, pseudo-code.
- One *worked example or diagram* to show more precisely how your algorithm works.
- A *proof* (or indication) of the correctness of the algorithm
- An *analysis* of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct solutions which are described clearly. Convoluted and obtuse descriptions will receive low marks.

In CS3230, you learn to develop high-level abstractions when describing algorithms. Try not to speak in ML/AL (machine/assembly language) or "for (j=0; j<n; j++) do". Instead give names to your sets (of objects or things or data structures), talk about Depth-First Search, Binary Search, traverse the graph, sort the set, use a priority queue, etc. You are no longer in CS1010, CS1020, CS2010 or CS2020. Speak with greater sophistication, and at a higher level of abstraction.

Solutions do not have to be typed-set, but they have to be neat and readable. Do not waste your time beautifying your answer sheet. If you want to spend more time, you should spend it on improving its clarity, understandability, and other aspects of its technical content.

Solving the A-Problems

The A-problems are *completely voluntary* and do not give more marks. They are meant for those who want to pursue algorithms in greater depth. Anyone can try any A-problem and turn in their serious solution attempts. If you obtain a solution with outside help (e.g., via collaboration, through the internet, through library work, some notes/textbook, other courses, etc), please acknowledge your source. In all cases, you should write up the solution on your own.

Please send your solutions/attempts on A-questions *separately* and directly to Prof. Leong. Do not attach them with the regular homework. I will look through them but I will not give them marks or grades. Instead, I give out "*coffee awards*" to good solutions attempts on A-problems. *Coffee awards* are claimable at any time, and have no expiry dates. We also take them into consideration only if you happen to be a borderline case (A-B border, B-C border, etc). There is no deadline for A-problems.

Advice of Solving CS3230 Homework Problems: (section added in Spring 2014)

You are advised to start on the homework *early*. Many students find algorithms design and analysis *difficult*. Very often a student will be able to understand an algorithm during the lectures, but may still find it hard to solve (or even get started on solving) homework problems. I believe you know that too. This is not unusual. Below are some advice; while they do not guarantee you a solution to your problems, they provide a good approach to finding a solution.

First, start with *understanding* the problem (what is the input, what properties it have, how is it organized/stored, what else do you know; what is the output required, what properties, etc).

Then work out some sample examples – start with simple example, then a bigger one, and a medium size one, until you know what is the input given and what is the output required. No need to work out any algorithm – just the input, and the required output.

Try to find a suitable model for the problem – Try out a graph model, a tree-based model, a database query model, etc to model your problem. Does it then map correctly to a well-known problem in these models? Is it a graph shortest path problem or a graph coloring problems, or a tree-search problem, etc. Will some standard known algorithm (that you have studied) work for this problem?

Explore different algorithmic strategies – start with those that you know for solving similar problems. For each of these, try out the algorithmic strategy on your examples (from above). You may have to create more examples that specifically test out your algorithm.

Be your own Devil's Advocate: Also, you have to play "devil's advocate" and try to find counter-examples that will "break your own algorithm". You may have to try many examples to achieve successfully find the counter-example, but it is worth the effort.

It may be *frustrating* that you can actually "*break your own algorithm*" and it may hurt your ego a bit, but that is actually OK and also an *excellent skill-set to develop* if you want to be good at algorithm design[1].

Iterate this process: By knowing what is wrong, you can "correct" your own algorithm (and call it version x+1). After successively breaking your own algorithms, you may finally come up with an algorithm that works! Of course, to make sure of that, you must make sure all the examples work out correctly and try as you may, you cannot come up with a counter-example. That's when you achieve the final "Aha" moment, and it is very satisfying!

Prove and Analyze your algorithm: Then you may need to prove that your algorithm is indeed correct. Then, of course, analyze its running time. To help in these, you may need the math that you learned long time ago, and maybe some of you have "*dismissed as useless*" – you know, like *calculus* (sequences, series, summation, L'Hopital rule), *discrete math* (sets, relations, functions, logic, proofs, recurrences, permutations/combinations, simple probability).

Look Back; Review; Look from a different perspective: Suppose you *have found* a solution, proved, and analyzed it, that's *good and very satisfying* and you really want to rush out and celebrate with an *i scream*. But, *WAIT*. . . if you do, then you'll miss the *best chance to really learn* and to do *meta-learning*. "*After you finish solving a problem, that's the best time for you to learn more about it and gain further insights into the solution.*"

Look back at your solution, and *ask*: What is the *key step* in the solution? Can I achieve this key step in another way? a faster way? Can I view the solution from a different perspective?

[1] The teaching staff of this course are very good at "*breaking algorithms*", a skill-set honed from grading many "*correct*" algorithms submitted to us over the course of many years of teaching.