**Worth:** 7.5% **Due:** Before 10pm on Tuesday 23 October.

**Remember to write the *full name*, *student number*, and *CDF/UTOR email address* of *each group member* prominently on your submission.**
*Please read and understand the policy on Collaboration given on the Course Information Sheet. Then, to protect yourself, list on the front of your submission **every** source of information you used to complete this homework (other than your own lecture and tutorial notes, and materials available directly on the course webpage). For example, indicate clearly the **name** of every student with whom you had discussions (other than group members), the **title** of every additional textbook you consulted, the **source** of every additional web document you used,* etc.

---

For each question, please write up detailed answers carefully. Make sure that you use notation and terminology correctly, and that you explain and justify what you are doing. Marks **will** be deducted for incorrect or ambiguous use of notation and terminology, and for making incorrect, unjustified, ambiguous, or vague claims in your solutions.

---

[12] 1. Consider nested function calls, such as $f(g(x), h(y, k()))$, and how they are evaluated by an interpreter. Suppose that, for technical reasons, functions require one time step for each nested function call they make, and only one function call can be made at each time step. Then the example expression above can be evaluated in four steps, as follows:

- time 0: call $f()$ — now we know that we have to call $g()$ and $h()$,
- time 1: $f$ calls $g()$,
- time 2: $f$ calls $h()$ — now we know that we have to call $k()$,
- time 3: $h$ calls $k()$,
- time 4: all done!

You should have no trouble convincing yourself that every order of calls will require the same number of steps (for example, if $f$ calls $h()$ first, it does not change the other calls that must be made, just their order).

Now suppose that we have access to a parallel computer that can execute multiple functions calls at the same time. It is still the case that functions require one time step for each nested function call they make — but now different functions can make calls at the same time. Then, the example above can be evaluated in only three steps — we cannot do better, because $f$ requires two time steps (one to call $h$ and one to call $g$):

- time 0: call $f()$ — now we know that we have to call $g()$ and $h()$,
- time 1: $f$ calls $h()$ — now we know that we have to call $k()$,
- time 2: $f$ calls $g()$ *and* $h$ calls $k()$, in parallel,
- time 3: all done!

This situation can be represented by the following abstract "Efficient Function Calls" problem.

**Input:** A nested function call expression $E$ of the form "$f(e_1, \ldots, e_k)$," where $f$ is a function name and each $e_i$ is either a variable or another nested function call expression (it is possible to have $k = 0$).

**Output:** The order of nested calls for each function in $E$, to minimize the total number of steps required to evaluate the expression on the parallel computer.

Give an algorithm to solve this problem efficiently: include a brief high-level English description, as well as a detailed pseudo-code implementation. Justify that your algorithm is correct, and analyze its worst-case running time.

[12] 2. A sawmill gets orders for different lengths of boards $\ell_1, \ell_2, \ldots, \ell_n$. All of the boards have to be cut from *planks*, long pieces of wood with a fixed length $L$, and for technical reasons, the boards have to be cut in the order they are given. No matter how the planks are cut into boards, there is usually some amount of waste left over from each plank.

For example, if the board lengths are $2, 4, 1, 5$ and $L = 7$, then we can cut the boards in many different ways — some of them are clearly silly:

- cut board 2 from one plank, 4 from another, 1 from another, and 5 from a fourth plank (leaving four waste pieces with lengths $5, 3, 6, 2$), or
- cut board 2 from one plank, 4 from another, and $1, 5$ from a third plank (leaving three waste pieces with lengths $5, 3, 1$), or
- cut board 2 from one plank, $4, 1$ from another, and 5 from a third plank (leaving three waste pieces with lengths $5, 2, 2$), or
- cut boards $2, 4$ from one plank, 1 from another, and 5 from a third plank (leaving three waste pieces with lengths $1, 6, 2$), or
- cut boards $2, 4$ from one plank and $1, 5$ from another (leaving two waste pieces of length 1 each), or
- cut boards $2, 4, 1$ from one plank and 5 from another (leaving only one waste piece of length 2).

Instructions like "cut boards $2, 5$ from one plank and $4, 1$ from another" are *not* valid solutions because they change the order of the boards.

From the point of view of the sawmill, what matters most is *not* how many planks are used, nor how much waste is left in total. What matters is that the waste pieces all be the same length, as much as possible, because this allows those pieces to be reused more easily for other purposes. So in the example above, the best choice is the second-last ($2, 4$ together and $1, 5$ together) — the last solution is not quite as good because the lengths of the waste pieces are 0 and 2 (the plank with no waste is considered to have waste of length 0).

Formally, consider the following "Board Cutting" problem.

**Input:** Board lengths $\ell_1, \ell_2, \ldots, \ell_n$ and plank length $L$, where each length is a positive integer, each $\ell_i \leqslant L$, and the board lengths are not necessarily all distinct.

**Output:** A division of $\ell_1, \ell_2, \ldots, \ell_n$ into groups $(\ell_{i_0+1}, \ldots, \ell_{i_1}); (\ell_{i_1+1}, \ldots, \ell_{i_2}); \ldots; (\ell_{i_{k-1}+1}, \ldots, \ell_{i_k})$, where $i_0 = 0$, $i_k = n$, $\ell_{i_j+1} + \ell_{i_j+2} + \cdots + \ell_{i_{j+1}} \leqslant L$ for $0 \leqslant j \leqslant k-1$ (intuitively: the total length of each group is no more than the length of one plank), and $\sum_{j=0}^{k-1} \left(L - \ell_{i_j+1} - \ell_{i_j+2} - \cdots - \ell_{i_{j+1}}\right)^2$ is minimized (intuitively: the lengths of the leftover pieces of plank are all as close to each other as possible).

Give a dynamic programming algorithm to solve this problem efficiently, following the steps shown in class. Include a brief high-level English description, as well as a detailed pseudo-code implementation. Justify that your algorithm is correct, and analyze its worst-case running time.

[20] 3. Suppose you are given a network $N$, a *maximum* flow $f$ on $N$, and one edge $e_0 \in N$ such that $f(e_0) = c(e_0)$. The flow $f$ is guaranteed to be maximum and to have integer flow values $f(e)$ for all edges $e$.

(a) If we *decrease* $c(e_0)$ by 1 (*i.e.*, let $c'(e_0) = c(e_0) - 1$), then we expect that the maximum flow on $N$ will also decrease by one unit. But does this always happen?

Either give a specific example where the maximum flow on $N$ does not change (and show that this is the case), or give a general argument that the maximum flow on $N$ always changes.

(b) Irrespective of your answer on the previous part, there are cases when this change in capacity causes the maximum flow to decrease.

Give an *efficient* algorithm that takes a network $N$, a maximum flow $f$ on $N$, and one edge $e_0 \in N$ such that $f(e_0) = c(e_0)$ and that determines a new maximum flow in the network $N'$, where $N' = N$ except for $c'(e_0) = c(e_0) - 1$.

Include a brief justification that your algorithm is correct (*i.e.*, explain how your algorithm works), and a brief analysis of your algorithm's worst-case runtime (which should be as small as possible).

(c) If we *increase* $c(e_0)$ by 1 (*i.e.*, let $c'(e_0) = c(e_0) + 1$), then we expect that the maximum flow on $N$ will also increase by one unit. But does this always happen?

Either give a specific example where the maximum flow on $N$ does not change (and show that this is the case), or give a general argument that the maximum flow on $N$ always changes.

(d) Irrespective of your answer on the previous part, there are cases when this change in capacity causes the maximum flow to increase.

Give an *efficient* algorithm that takes a network $N$, a maximum flow $f$ on $N$, and one edge $e_0 \in N$ such that $f(e_0) = c(e_0)$ and that determines a new maximum flow in the network $N'$, where $N' = N$ except for $c'(e_0) = c(e_0) + 1$.

Include a brief justification that your algorithm is correct (*i.e.*, explain how your algorithm works), and a brief analysis of your algorithm's worst-case runtime (which should be as small as possible).