NOTE TO STUDENTS: This file contains sample solutions to the term test together with the marking scheme and comments for each question. Please read the solutions and the marking schemes and comments carefully. Make sure that you understand why the solutions given here are correct, that you understand the mistakes that you made (if any), and that you understand *why* your mistakes were mistakes.

Remember that although you may not agree completely with the marking scheme given here it was followed the same way for all students. We will remark your test only if you clearly demonstrate that the marking scheme was not followed correctly.

For all remarking requests, please submit your request **in writing** directly to your instructor. For all other questions, please don't hesitate to ask your instructor during office hours or by e-mail.

GENERAL MARKER'S COMMENTS:

- Pay a little attention when using notation referring to sets as opposed to sequences.
- When you use recursion to describe a DP algorithm you should be very careful to memoize it. Otherwise you end up running in exponential time.
- Induction is not magic. You have an inductive claim (induction predicate) on the natural numbers and you perform induction to show it's true for every natural number. Showing the induction basis and then defining the predicate makes no sense. You don't prove the induction basis for the induction basis. You don't follow the steps of the induction just to get full marks! :) You **start** by defining the induction claim (predicate) and then you do all the required steps.
- Also, it doesn't make any sense to do induction without using the induction hypothesis in the construction of your inductive claim. You have something from your hypothesis. Take this "something" and construct "something new" in your induction step. Mention all these things explicitly.

The following comment codes were used. The fact that a comment code appears in your paper does not mean that you necessarily lost marks because of it (depends on the context of your writeup for the specific question). Refer to the marking scheme for each question for more specific details.

- **E1:**   What are you proving by induction? Induction on what? (You may see this comment for any of these two reasons.)
- **E2:**   Define promising solution explicitly.
- **E3:**   No justification/not sufficient justification for the recurrence relation.
- **E4:**   State the recurrence explicitly.
- **E5:**   Compute the actual path, not only its length.
- **E6:**   Where do you use the induction hypothesis?
- **E7:**   Running time?
- **E8:**   The definition of a promising solution has nothing to do with the iterations of the algorithm. The details of the algorithm should not appear in the definition of promising solution.
- **E9:**   Extend the solution to optimal using what?

Question 1.   [14 MARKS]

You are consulting for a trucking company that does a large amount of business shipping packages from Toronto to Montréal. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed integer limit $W > 0$ on the maximum amount of weight they are allowed to carry. Boxes arrive at the Toronto loading platform one by one, and each package i has a positive integer weight $w_i \leqslant W$. The loading platform is quite small, so at most one truck can be at the platform at any time. Company policy (and the limited space available at the platform) requires that boxes are shipped in the order they arrive—there would be complaints if boxes made it to Montréal out of order!

At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive (as required), putting as many boxes as possible in each truck without exceeding the weight limit W, *i.e.*, whenever the current box would make the load exceed the weight limit W, they send the truck on its way and keep the box for the next truck. But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved: maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed? More formally, the current algorithm used by the company can be written down as follows.

> TRUCKPACKING($W, w_1, w_2, \ldots, w_n$):
> $k = 1$ # current truck number
> $T_k = [w_1]$ # list of packages in current truck
> $t = w_1$ # total weight of current truck
> $S = [T_k]$ # current partial solution
>   **for** $i = 2, \ldots, n$:
>   **if** $t + w_i \leqslant W$:   # package i fits in current truck; put it in
> T_k.append(w_i)
> $t += w_i$
>   **else**:   # package i does not fit in current truck; start next one
> $k += 1$
> $T_k = [w_i]$
> $t = w_i$
> S.append(T_k)
>   **return** S # $= [T_1, T_2, \ldots, T_k]$

Following the structure given in class, give a detailed proof that the current algorithm is guaranteed to always use the minimum number of trucks. In particular, give a clear, precise definition of what it means for a partial solution to be "promising" for this problem. (Your solution will be marked on its structure as well as its content.)

SAMPLE SOLUTION:

> Let $S_1, S_2, \ldots, S_n$ be the sequence of partial solutions generated by the algorithm, at the end of each iteration of the main loop (*i.e.*, S_i has put packages $1, 2, \ldots, i$ into trucks, but has not yet considered package $i + 1$). We say that S_i is "promising" if there is an optimal solution $\hat{S}_i$ such that $\hat{S}_i$ puts packages $1, 2, \ldots, i$ on the same trucks as S_i. We prove that $S_1, S_2, \ldots, S_n$ are all promising, by induction.
>
> **Base Case:** S_1 is promising because every optimal solution must put package 1 on the first truck, which is what the algorithm does.

Ind. Hyp.: Suppose $i \in \{1, 2, \ldots, n\}$ and S_i is promising, *i.e.*, there is an optimal solution $\hat{S}_i$ that puts packages $1, 2, \ldots, i$ on the same trucks as S_i.

Ind. Step: Consider S_{i+1}. Either package $i+1$ is put on a new truck, or it is put on the same truck as package i (call this truck T_j).

     Case 1: If package $i+1$ is put on a new truck (T_{j+1}), it must be because w_{i+1} plus the total weight of the packages already on truck T_j exceeds the weight limit W. Hence, $\hat{S}_i$ cannot put package $i+1$ on truck T_j, which means it must be put on truck T_{j+1} (because packages cannot be send out of order). So $\hat{S}_{i+1} = \hat{S}_i$ puts packages $1, 2, \ldots, i+1$ on the same trucks as S_{i+1}.

     Case 2: If package $i+1$ is put on truck T_j (the one that contains package i), then either $\hat{S}_i$ puts package $i+1$ on truck T_j, or it doesn't.

         Subcase A: If $\hat{S}_i$ puts package $i+1$ on truck T_j, then $\hat{S}_{i+1} = \hat{S}_i$ puts packages $1, 2, \ldots, i+1$ on the same trucks as S_{i+1}.

         Subcase B: If $\hat{S}_i$ puts package $i+1$ on a different truck, it must be on truck T_{j+1} (because packages cannot be send out of order). Let $\hat{S}_{i+1}$ be the same as $\hat{S}_i$ except that package $i+1$ is moved from truck T_{j+1} to truck T_j. Then, $\hat{S}_{i+1}$ is an optimal solution (it uses no more trucks than $\hat{S}_i$ and is a valid solution because package $i+1$ fits on truck T_j) that puts packages $1, 2, \ldots, i+1$ on the same trucks as S_{i+1}.

     In every case, S_{i+1} is promising.

Hence, $S_1, S_2, \ldots, S_n$ are all promising. In particular, S_n is promising, which means that there is an optimal solution $\hat{S}_n$ that puts packages $1, 2, \ldots, n$ on the same trucks as S_n—but this is simply saying that S_n is optimal.

MARKING SCHEME:

- **Structure:** [3 marks]
  clear attempt to define "promising", to prove each partial solution is promising (by induction on the number of iterations of the main loop), and to use this to conclude that the greedy solution is optimal
- **Promising:** [3 marks]
  correct definition of "promising"—give up to 2 marks if the notion is used correctly, even if it was not defined explicitly at the start
- **Induction:** [4 marks]
  correct proof by induction, up to the exchange argument (including setting up the correct cases and sub-cases)
- **Exchange:** [2 marks]
  correct proof of the exchange argument
- **Conclusion:** [2 marks]
  correct proof that the final solution is optimal

Question 2.   [19 MARKS]

An "ordered graph" $G = (V, E)$ is a **directed** graph whose nodes can be ordered $v_1, v_2, \ldots, v_n$ such that:

(i) every edge goes from a lower indexed node to a higher indexed node, *i.e.*, $i < j$ for all $(v_i, v_j) \in E$;

(ii) all nodes except v_n have at least one outgoing edge.

Given an ordered graph $G = (V, E)$, we want to find a longest path from v_1 to v_n, *i.e.*, a path with the maximum number of edges. For example, in the graph below, (v_1, v_2), (v_2, v_4), (v_4, v_5) is a longest path of length 3.
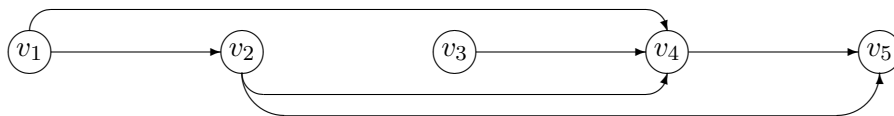
Part (a)   [5 MARKS]

Give a simple greedy algorithm that attempts to solve this problem, on input $G = \big(V = \{v_1, v_2, \ldots, v_n\}, E\big)$ where the vertices have already been ordered to satisfy the conditions above. Then, show that your algorithm does not always return an optimal solution (give an explicit counter-example and describe the solution returned by your algorithm, as well as why it is not optimal).
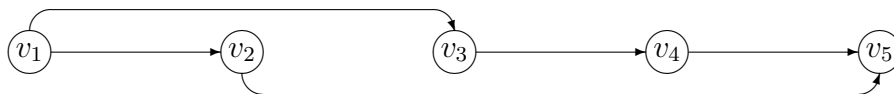
SAMPLE SOLUTION:

> Main idea of algorithm: take edges to low-numbered vertices as much as possible, to stretch out the path.

> LONGPATH($G = (V, E)$):
> $P = [\,]$ # partial path found so far
> $i = 1$ # current vertex
>   **while** $i < n$:
>   find smallest j such that $(v_i, v_j) \in E$
> $P = P + \big[(v_i, v_j)\big]$
> $i = j$
>   **return** P

> Counter-example:

> The algorithm returns path $P = \big[(v_1, v_2), (v_2, v_5)\big]$ of length 2, when the graph contains a path $\big[(v_1, v_3), (v_3, v_4), (v_4, v_5)\big]$ of length 3.

MARKING SCHEME:

- **Structure:** [1 marks]
  clear attempt to give an explicit greedy algorithm and to provide an explicit counter-example for the algorithm
- **Algorithm:** [2 marks]
  reasonable and clear greedy algorithm—give full marks even if the algorithm is not given in pseudo-code, as long as the description is clear and detailed enough
- **Counter-example:** [2 marks]
  good counter-example, including justification that the algorithm fails

Part (b)   [14 MARKS]
Following the structure given in class, give a dynamic programming algorithm for this problem; state the running time of your algorithm. (Your solution will be marked on its structure as well as its content.)

SAMPLE SOLUTION:

> For $i = 1, 2, \ldots, n$, let $L[i]$ be the length (number of edges) of a longest path from v_1 to v_i ($L[i] = -\infty$ if there is no path from v_1 to v_i).
>
> Then, $L[1] = 0$ because the only path from v_1 to v_1 is the one with no edge, and for $i = 2, \ldots, n$,
> $$L[i] = \max \left\{ 1 + L[j] : (v_j, v_i) \in E \right\}$$
> ($L[i] = -\infty$ if there is no edge $(v_j, v_i) \in E$), because a longest path from v_1 to v_i must consist of a longest path from v_1 to v_j followed by the edge (v_j, v_i), for some $j < i$.
>
> The values in L can be computed in a bottom-up fashion, as follows (the meaning and purpose of array P will be explained below):

> $L[1] = 0$
> $P[1] = 0$
> **for** $i = 2, \ldots, n$:
> $L[i] = -\infty$
> $P[i] = -1$
>   **for** $(v_j, v_i) \in E$:
>   **if** $1 + L[j] > L[i]$:
> $L[i] = 1 + L[j]$
> $P[i] = j$

> To reconstruct the actual longest paths, we use a second array $P[i]$ to store the predecessor of i on a longest path from v_1 to v_i, while computing the values of L. Then, the following code constructs a longest path from v_1 to v_n:

> $P = []$ # current path
> **if** $L[n] > -\infty$:   # if there is a path from v_1 to v_n
> $i = n$ # current vertex
>   **while** $i > 1$:
> $P = \left[(v_{P[i]}, v_i) \right] + P$
> $i = P[i]$
> # At this point, P is a longest path from v_1 to v_n.

> The running time of the first piece of code is $\Theta(n^2)$ (because of the nested loops) and for the second piece of code it is $\Theta(n)$, for a total of $\Theta(n^2)$.

MARKING SCHEME:

- **Structure:**   [3 marks]
  clear attempt to define an array, give a recurrence relation for it (with justification), compute its values bottom up, use those values to construct a solution, and state the running time of the entire algorithm
- **Definition:**   [1 mark]
  reasonable and clear array definition

- **<u>R</u>ecurrence:**   [3 marks]

  correct recurrence for the array, including appropriate base case(s)
- **<u>J</u>ustification:**   [2 marks]

  good justification for the recurrence—give part marks even if the recurrence is incorrect, for reasonable attempts to justify it
- **<u>B</u>ottom-up algorithm:**   [1 mark]

  correct bottom-up algorithm to compute array—give full marks for correctly computing array values following the recurrence, even if array and/or recurrence are incorrect
- **<u>C</u>onstructing solution:**   [3 marks]

  correct algorithm to find actual solution, including technique used to figure out correct predecessor (either a second array or an appropriate loop)—give full marks for correctly finding a solution assuming that the array values have been computed correctly, even if they haven't been
- **<u>T</u>ime:**   [1 mark]

  correct runtime stated for the algorithm