COMP3121/9101
Algorithm Design

Tutorial 3
The Greedy Method

Before the Tutorial

Before coming to the tutorial, try and answer these discussion questions yourself to get a firm understanding of how you're pacing in the course. No solutions to these discussion points will be officially released, but feel free to discuss your thoughts on the forum. Your tutor will give you some comments on your understanding.

- There are primarily two characteristics of an algorithm that makes it *greedy*. Briefly outline how the Activity Selection problem from lectures maintain these characteristics.
  - **Greedy choice property**: At each choice, we always make the best local choice.
  - **Optimal substructure property**: The solution to each subproblem depends on the solutions of previously solved subproblems (i.e. we are always iterating over local optimal solutions to obtain the final global optimal solution).
- Greedy algorithms are often simple to describe because they rely on local optimal solutions to provide the global optimum. It still remains to prove that our greedy algorithm is correct.
  - In this tutorial sheet, we look specifically at one method of proof – greedy stays ahead.
  - Briefly outline the *greedy stays ahead* method of proof. You may want to consider how we perform inductive proofs.
  - Why does the *greedy stays ahead* method prove that a greedy algorithm is *optimal*?
- Read through the problem prompts and think about how you would approach the problems.

Tutorial Problems

Problem 1. You are driving along one road from Sydney to Perth and you can only drive 100km in a day. You have a list of n hotels sorted by its distance from Sydney and you need to ensure that you stop at a hotel every night. Ideally, you'd like to stop for the fewest number of nights possible – how should you plan out your trip?

(a) Devise a greedy strategy to solve the problem.

(b) Justify the correctness of your algorithm, using a "greedy stays ahead" proof.

- Think carefully about how you should measure two solutions.
  - What aspect of your algorithm are you trying to maximise or minimise?
- Let $\mathcal{G} = (g_1, \ldots, g_m)$ denote our greedy solution and $\mathcal{O} = (o_1, \ldots, o_{m'})$ be *any* optimal solution.

  - Do we actually require $\mathcal{O}$ to be an optimal solution?
  - How should we define g_i and o_i?
  - Why do we index $\mathcal{G}$ by m and $\mathcal{O}$ by m'?
- Show that the base case holds.
  - You will need to show that either $g_1 \leq o_1$ or $g_1 \geq o_1$ depending on how you define your heuristic.
- Assume that the greedy solution stays ahead for all $\ell < k$ (i.e. for all $\ell < k$, either $g_\ell \leq o_\ell$ or $g_\ell \geq o_\ell$). Show that the greedy solution also stays ahead at the kth iteration (i.e. $g_k \leq o_k$ or $g_k \geq o_k$).
- Hence, conclude that your algorithm is correct.

(c) Analyse the time complexity of your algorithm.

Problem 2. Recall the problem from Tutorial 1, Problem 2.

We have to deliver n packages to a town within K days. Each package i has an associated weight w_i and we can only load the packages into a truck in the order of their position on the conveyor belt. We may not load more than the capacity of the truck.

(a) Given the capacity C of the truck, the weights of the truck, and the number of days K, devise a greedy algorithm that determines whether the truck can deliver all n packages to the town within the K days.

(b) Justify the correctness of the algorithm, using a "greedy stays ahead" proof.

- Imitate the scaffold from Problem 1.

After the Tutorial

After your allocated tutorial (or after having done the tutorial problems), review the discussion points. Reflect on how your understanding has changed (if at all).

- In your own time, try and attempt some of the practice problems marked [K] for further practice. Attempt the [H] problems once you're comfortable with the [K] problems. All practice problems will contain fully-written solutions.
- If time permits, try and implement one of the algorithms from the tutorial in your preferred language. How would you translate high-level algorithm design to an implementation setting?