# COMS21103 - Coursework 1

Ross Gardiner

November 26, 2015

2 a)

$$\text{P}[i,x] = \begin{cases} 0 & \text{if } i = 0 \\ max(P[i-1,s]-m, P[i-1,u]) + U[i] & \text{if } x = u \\ max(P[i-1,s], P[i-1,u]-m) + S[i] & \text{if } x = s \end{cases}$$

2 b) The correctness of this formula can be shown by induction. Informally, after 0 days, the profit must be 0. After $> 0$ days, the profit will be the (max profit from the current day)+(the profit from the previous days).

The scenario in which a greedy algorithm would fail is where it has to choose whether to switch or stick without knowing the future impact of that decision.

Because the recursive algorithm relies on the pre-calculated maximum prior profit, it will only switch stock if the switch increases takings by at least the cost of switching plus the takings from the other stock type. As a result, it will never end up in a situation where it switches stock inefficiently.

2 f) TODO

2 a) $\textsc{Magic-Tournament}[magician] = \textsc{Maximum}\ \big[\text{M}[magician, true],\ \text{M}[magician, false]\big]$
(where $magician$ is the head magician)

Let:
$m \equiv$ root magician of some subtree
$s \equiv$ is the root magician included?

Using Haskell-ish notation:

$$\text{M}[m,s] = \begin{cases} (\textsc{Fold}\ (+)\ (\textsc{Map}\ (a \mapsto \text{M}[a, false])\ m.apprentices)) + m.ability & \text{if } s \\ \textsc{Fold}\ (+)\ (\textsc{Map}\ (a \mapsto (\textsc{Maximum}\ \big[\text{M}[a,true],\ \text{M}[a,false]\big]))\ m.apprentices) & \text{if } \neg s \end{cases}$$

2 b) The algorithm explores every possibility. Each magician in the tree is at the root of his own tree—possibly empty. M can be called such that it either includes the root magician or excludes the root magician. In the former case, all of the apprentices must be excluded. In the latter case, the algorithm finds the maximum result from either including or excluding each apprentice.

At the root of the tree, we try calling M both including and excluding the head magician, and pick the highest result.

2 f) TODO

3 d) There is one use for memoization. When `unselected()` calls `selected()` and `unselected()`, they both call `unselected()` on the same set of apprentices. The results of the calls to `unselected()` could be cached.

4 a) $\text{S}[f,r]$ returns the sandwiches deliverable from day $f$ onwards given that the last day off was $r$ days ago. The inital call is $\text{S}[1,0]$.

$$\mathrm{S}[f,r] = \begin{cases} 0 & \text{if } f > n \\ min(B[f], M[r]) & \text{if } f = n \\ min(B[f], M[0]) + \mathrm{S}[f+1, 1] & \text{if } r = 0 \\ max(min(B[f], M[r]) + \mathrm{S}[f+1, r+1], \mathrm{S}[f+1, 0]) & \text{otherwise} \end{cases}$$

4 b) If the start day is after the last day, no sandwiches can be delivered. If the start day is the last day, the most sandwiches deliverable from that day is the number that can be delivered on that day. These are the base cases.

Otherwise, if the previous day was a rest day ($r = 0$), the only option that makes sense is to deliver on the start day—two consecutive rest days is never optimal. This is then added to the maximum number of sandwiches deliverable from the next day (a recursive call).

Finally, it the previous day was not a rest day, we must find the maximum result from two options: either taking a rest day and setting $r = 0$ or delivering on the start day and continuing to increment $r$.

4 c) $\mathrm{S}[1, 0]$

4 f) TODO

4 g) Mostly converted to tail call recursion (`IterativeMostSandwichesDeliverableCalculator::sandwiches`) and iteration (`IterativeMostSandwichesDeliverableCalculator::sandwichesIterative`), but can't figure out how to handle the multiple recursion in `max()`.

4 h) TODO