# CS 344 Homework Three

**Matthew McCaughan**

Due March 27th, 2025

**Problem 1: Academic Integrity**

I affirm that I have not given or received any unauthorized help on this assignment, and that this work is my own.
- Matthew McCaughan

**Problem 2: Stacking Boxes**

**Algorithm:** We're going to be utilizing a greedy approach. Sort the boxes by greatest $l_i$, the maximum weight the box can tolerate on top of it without being crushed. The first box (bottom of the stack) will be the box with the greatest $l_i$. For every subsequent box on the stack after the first is selected, ensure that the sum of the weights from the current box to the top of the stack is less than or equal to the carrying capacity of the box below.

$$\sum_{current}^{top} w_i \leq l_{belowcurrent}$$

Each box can have an attribute that tracks this weight. Do this for every box as you add them. If at any point a box cannot support the weight above it with the addition of a box, then no such solution will exist.
**Proof of Correctness:** Using an exchange argument: Assuming another algorithm $ALG$ has correctly found a solution in conjunction with our greedy algorithm. $ALG$ has not sorted the boxes by $l_1$ and applies an arbitrary method to stack. At the first box in the stacks that differ between these two algorithms, we can assume that for this box, $l_iALG < l_igreedy$ because greedy will select the box with greatest available $l_i$, so any difference (unless there are duplicate value boxes) will result in a lesser $l_i$ selected by $ALG$. This means that $ALG$ may select a box where the box above will have a greater $l_i$ than the selection. This can lead to a non-ideal stacking where boxes that can support greater weights are higher up on the stack, and potentially lead to a 'non-feasible' solution where one in-fact exists. If we replace this box from $ALG$ to the one selected by our greedy algorithm to produce a new order $ALG'$, we guarantee that the greatest weight can be supported at every level by the definition of the greedy algorithm, and the weight that can be supported is at least as great as the one selected by $ALG$, leading to a valid solution every time one exists.
**Run Time:** It would take at least in the order of $O(n*log(n))$ to sort the boxes by decreasing $l_i$, and we iterate through the number of boxes $n$ while checking that cumulative weights to not exceed the supported weights of the boxes. Assuming that we can update the cumulative weights by instantly accessing and modifying the values associated with the boxes in the stack, then our algorithm can be performed in constant time. This cumulatively gives us a running time within the order of $O(n * log(n) + n)$, or more simply, $O(n * log(n))$.

**Problem 3: Not all meetings are the same**

Much like our meeting scheduling problem from class, a good starting point is sorting the meetings by end time $e_i$. Instead of selecting our meetings by this explicit metric, we could potentially get a higher importance score from fewer meetings, so our decisions must be made at every opportunity with respect to the importance score $I(i)$ we want to maximize. For every meeting in our order of end time, we will either attend the meeting or we wont. If we attend the meeting, its our current score up to this point plus the score we get from attending the meeting. If we do not attend the meeting, its our score up to and excluding this meeting, while making sure we can actually fit the meeting. We can define a function $M(m_i)$ which gives the maximum score including meetings (previously sorted) from index 0 to index i:

$$M(Meet_i) = \begin{cases} 0, & \text{if M(0) (base-case)} \\ \text{max(} & \\ \ \text{M}(Meet_i - 1), & \\ \ \text{M}(Meet_i - 1) + I(i) \text{ \&\& Meet is Available} & \\ \ \text{)}, & \text{otherwise} \end{cases}$$

The "Meet is Available" component can either be a boolean function that must be satisfied in order to be considered (else other condition will always be greater), or it can return a value that does not affect the score when the meeting is available but returns a really negative value if the meeting is not available to emulate the boolean. If you make this check in constant time with array accessing the current meeting and previous meetings' $(s_i, e_i)$ and performing a check, then our most significant components to the runtime analysis include the sorting the iteration through every actual meeting.
**Runtime:** Sorting the meetings in order of end time will require at least in the order of $O(n * log(n))$, and then we will iterate through the sorted list of meetings once and perform our decision at every point. With no need to recalculate previous values and instant access to meeting scores, this will take within the order of $O(n)$. Combining these components to get our overall running time within the order of $O(n * log(n) + n) = O(n * log(n))$

**Problem 4: Minimum Ice Cream Delay**

**Algorithm:** We're going to be utilizing a greedy approach. Based upon our generous hint, we want to minimize the number of times we have to run back to the fridge, and we will do that by choosing at every instance we need to replace, the flavor that will go the longest time without an order to go back to the fridge. Else, if the flavor is available to serve, serve the flavor. This decision making process will result in the lowest number of times we will have to run back to the fridge to swap a flavor.
**Proof of Correctness:**
Using an exchange argument: Assuming another algorithm $ALG$ has correctly found a solution in conjunction with our greedy algorithm. $ALG$ is employing some different strategy and replaces a flavor arbitrarily from the front when needed. At the first replacement of the flavors that differ between the two algorithms, we can assume that for the flavor $F$ being replaced, The next request for F will occur at the same time or father in the future for the Greedy Algorithm than it does for $ALG$, since our greedy algorithm will be guaranteed to replace the farthest requested flavor. This also means that this flavor will be brought back to

the front at some time $t$ for $ALG$ that is greater than or equal to the greedy algorithm. That means within this time frame, we will be at worst spending one more swap in $ALG$ than what we would be for our greedy algorithm. We can replace the swap we make for $ALG$ for the swap we make in our greedy algorithm. This replaces 1 swap for 1 swap, so our performance cannot worsen, and we will not have to perform a next swap at an earlier time, which is likely to improve performance by decreasing the number of swaps overall.
**Run Time:** For each request within our time frame $T$, we are checking the $k$ different flavors at the front, and each flavor looks ahead into the description list, which is at worst $T$ steps ahead. We will get an overall run time within the order of $O(T^2 * k)$. Your performance is interestingly not dependent on the number of flavors, just how many you can store in the front.

**Problem 5: Binary Tree Equivalency**

**(a):** Assuming we knew a pairing $r_1$ to $r_2$ between the two trees. We can perform two Breadth First Searches (one for each vertex). Subsequent matching vertices will be at the same depths and have the same color based on the BFS trees for these starting vertices. We know that no two vertices that are siblings will have the same color. With constant time attribution of vertices, our only time sink is our two BFS searches, on the order of $O(2 * (V + E))$, or within our $O(n)$ constraint.
**(b):** To find two vertices that are as far apart as possible given a tree, perform a Breadth First Search on a vertex with the greatest depth. This node is guaranteed to be included in the greatest distance. With just a BFS from a known vertex to perform, this can be completed within the order of $O(V + E)$, or within our $O(n)$ constraint.
**(c):** With multiple pairs of vertices that are as far apart as the farthest pair, the midpoint of the path between or the two midpoints will always be the same. The vertices connected to the midpoint(s) will either be a parent or child of this vertex, in either case they will have the same depth difference to this vertex, which will be 1. Since the farthest pairs of vertices contain a vertex that is at the lowest depth, multiple pairs of longest paths will all have vertices at the lowest depth; since these vertices are on the same depth, they will have the same distance to the midpoints and will meet at the same midpoint.
**(d):** To composite together an algorithm that decides if two trees $T_1$ and $T_2$ are the same, we can find a pair of vertices that are as far apart as possible, and get the midpoint of the pair. We can perform BFS on that pair and compare the subsequent BFS trees for both trees. If these match and every vertex on $T_1$ can be mapped to a vertex on $T_2$, then these trees will be equivalent. And since we are only performing consecutive BFS searches in our algorithm, our running time will be within the order of $O(n)$.

**Problem 6: Digraph walking**

**(a):** Assuming that $G$ a strongly connected graph, we can reach any other vertex from any other vertex including our starting vertex $v$, giving us essentially no limitations. Since we can really do whatever we want we can just go to each vertex at some point and collect its score. If we need to explicitly walk to each vertex, then we can perform a simple BFS/DFS search and add the score to an aggregate, else if we are omniscient of the scores we do not need to actually perform the walk. We can just take all the scores provided and sum them

up. At worst, a BFS/DFS search will run within the order of $O(m+n)$
**(b):** Assuming that our graph $G$ is a DAG, we are more limited, and we can leverage DFS to give us a topological sorting of the graph. Since no cycles exist in a DAG, then DFS(G) will implicitly construct a DFS forest for the vertices. We may hit previously explored trees but if in our tree $T$ we come along the root of another tree in the forest $T'$ we can simply place $T'$ downstream of $T$. This can be repeated until we are left with one tree, and the root of our full tree is where we should start. While traversing the tree in DFS, we can update edge weights recursively as we return from vertices, so we can get a heuristic as to maximizing our score when we subsequently start from our topological root vertex. Since DFS runs within the order $O(m+n)$ we are within our constraint.
**(c):** For general digraphs, we wont know whether or not a cycle will exist or if the whole graph is strongly connected, so before anything, we can perform Kosaraju's algorithm to identify the strongly connected components, and then perform the DFS topological search among these strongly connected components. So that we can maximize our score among the SCC, by making a DAG of SCCs, or move through the whole tree in the case that G is strongly connected. Kosaraju's algorithm runs with two iterations of DFS, we can perform our general digraph algorithm within the other of $2 * O(m+n) = O(m+n)$.