Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Riya Kore Wisc id: rykore

## More Greedy Algorithms

1. *Kleinberg, Jon. Algorithm Design (p. 189, q. 3).*

   You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit $W$ on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package $i$ has a weight $w_i$. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

   Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Hint: Use the stay ahead method.

You can use induction to prove that the given algorithm minimizes the no. of trucks needed. If you perform induction on the number of trucks, we can prove that this strategy is efficient.

Base case: 1 truck

When you have one truck, you can fit only what can be packed inside.

Inductive hypothesis: Assume that this algorithm works for k trucks.

Inductive step: When we have k+1 trucks

By the inductive hypothesis, we know that stay ahead has shipped at least as many boxes as any other strategy/algorithm S in the first k trucks.

For the other strategies S to exceed stays ahead with the $(k+1)^{st}$ truck, S would have to pack all the items packed by stays ahead in its $(k+1)^{st}$ truck plus the next item j. But, according to the base case, this is not possible as, with one truck, only what fits can be packed. Strategy S cannot just add an extra box as it wont be possible. This means that the $(k+1)^{st}$ truck for the strategy S and stays ahead will both carry the same amount of boxes proving that for a given set of boxes with specific weights, the greedy algorithm currently in use actually minimizes the no. of trucks that are needed.

2. *Kleinberg, Jon. Algorithm Design (p. 192, q. 8).* Suppose you are given a connected graph $G$ with edge costs that are all distinct. Prove that $G$ has a unique minimum spanning tree.

We need to prove that the connected graph $G$ has one unique minimum spanning tree. Let's prove this using proof by contradiction. For this, let's assume that $G$ has at least 2 minimum spanning trees, $N_1$ and $N_2$ that are different.

Let $e \in N_1 \setminus N_2 \cup N_2 \setminus N_1$ be the minimum cost edge not shared by $N_1$ and $N_2$.

WLOG, lets assume that $e$ comes from $N_1$. We can create another graph $N_2 \cup e$ which now contains a cycle $C$ having $e$ in the cycle. Let $f$ be the most expensive edge in $C$ / the edge having the highest weight.

If $f = e$, then $N_1$ is not a minimum spanning tree (Let $C$ be any cycle in a connected graph $G$, and $e$ be the most expensive edge of $C$. Then, $e$ is not in any minimum spanning tree of $G$).

If $f \neq e$, then let $N_3 = N_2 \cup e \setminus f$.

$N_3$ is a tree and $N_3$ must have a lower cost than $N_2$ since $c_e < c_f$ and they cannot be equal because we have assumed that each edge weight is unique.

This is a contradiction as the overall cost of $N_3$ is strictly less than $N_2$. Our initial assumption that $G$ has at least 2 minimum spanning trees is false. This means that $G$ has only one unique minimum spanning tree.

(vertice, edge)

3. *Kleinberg, Jon. Algorithm Design (p. 193, q. 10).* Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree $T$ in $G$. Now assume that a new edge is added to $G$, connecting two nodes $v, w \in V$ with cost $c$.

   (a) Give an efficient ($O(|E|)$) algorithm to test if $T$ remains the minimum-cost spanning tree with the new edge added to $G$ (but not to the tree $T$). Please note any assumptions you make about what data structure is used to represent the tree $T$ and the graph $G$, and prove that its runtime is $O(|E|)$.

For the graph G, let's represent G using an adjacency list.
Each adjacent node will have a cost of the edge and a bit associated with it. If the associated bit is 1, then that edge belongs to the minimum spanning tree.

For the algorithm:

1) Start from v and do a depth first search on T until you find w.
2) As you go, you store the simple path from v to w path.
3) Look at the cycle C formed by adding the new edge. If the cost of the new edge is not the maximum cost edge of C, then T is no longer a minimum spanning tree.

To check the runtime of this algorithm:

- for depth first search : $O(|V| + |E|) = O(|E|)$.
- checking the maximum cost of the path from v to w: $O(|E|)$.

   (b) Suppose $T$ is no longer the minimum-cost spanning tree. Give a linear-time algorithm (time $O(|E|)$) to update the tree $T$ to the new minimum-cost spanning tree. Prove that its runtime is $O(|E|)$.

Algorithm:

1) Start from v and do a depth first search on T until you find w.
2) As you go, store the path from v to w.
3) Look at the cycle c formed by adding the new edge, and let f be the most expensive edge.
4) Update the minimum spanning tree bit for f to 0, and update the minimum spanning tree bit for the new edge added to 1.

To check for the runtime of this algorithm:

- depth first search : $O(|V| + |E|) = O(|E|)$.
- to find the max cost edge of the v to w path: $O(|E|)$.
- changing the bits of the minimum spanning tree to add in the new edge : $4 \times O(|E|)$.

4. In class, we saw that an optimal greedy strategy for the paging problem was to reject the page the furthest in the future (FF). The paging problem is a classic online problem, meaning that algorithms do not have access to future requests. Consider the following online eviction strategies for the paging problem, and provide counter-examples that show that they are not optimal offline strategies.[1]

   (a) FWF is a strategy that, on a page fault, if the cache is full, it evicts all the pages.

counter-example :
Request sequence : $\sigma = \langle a, b, c, a, c \rangle$
cache size : $k = 2$

for FWF: After the first two requests, the cache becomes full and FWF will evict the entire cache causing 2 more page faults. overall, the page fault count will come to 3.

for FF: After the first two requests, the cache is full and FF will evict b to bring c with no page fault on the last request. overall, the page fault count will come to 2.

   (b) LRU is a strategy that, if the cache is full, evicts the least recently used page when there is a page fault.

counter-example:
request sequence: $\sigma = \langle a, b, c, a, b \rangle$
cache size : $k = 2$

① a, b
② a, c
③ a, b

for LRU: After the first two requests, the cache is full and LRU will evict a to bring in c. Then, it will evict b to bring in a. Lastly, it will evict c to bring in b. overall, there will be 4 page faults.

for FF: After the first two requests, the cache is full and FF will evict b to bring in c. Then, it will evict c to bring in b with no page fault on the last request. Overall, there will be 3 page faults.

[1] An interesting note is that both of these strategies are $k$-competitive, meaning that they are equivalent under the standard theoretical measure of online algorithms. However, FWF really makes no sense in practice, whereas LRU is used in practice.

# Coding Problem

5. For this question you will implement Furthest in the future paging in either C, C++, C#, Java, Python, or Rust.

   The input will start with an positive integer, giving the number of instances that follow. For each instance, the first line will be a positive integer, giving the number of pages in the cache. The second line of the instance will be a positive integer giving the number of page requests. The third and final line of each instance will be space delimited positive integers which will be the request sequence.

   Note: a naïve solution doing repeated linear searches will timeout.

   A sample input is the following:

```
3
2
7
1 2 3 2 3 1 2
4
12
12 3 33 14 12 20 12 3 14 33 12 20
3
20
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

   The sample input has three instances. The first has a cache which holds 2 pages. It then has a request sequence of 7 pages. The second has a cache which holds 4 pages and a request sequence of 12 pages. The third has a cache which holds 3 pages and a request sequence of 15 pages.

   For each instance, your program should output the number of page faults achieved by furthest in the future paging assuming the cache is initially empty at the start of processing the page request sequence. One output should be given per line. The correct output for the sample input is

```
4
6
12
```