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: Zinnia Nie Wisc id: 908 319 4044

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.

Solution: Greedy: Pack boxes in order until reach max w a truck can carry

Stays Ahead Greedy: $G = g_1, g_2, \ldots, g_n$ } are boxes that are fit in
Optimal: $S = s_1, s_2, \ldots, s_m$ } the first k trucks where $n \geq m$

Proof by Induction:

Base case: k=1, in the greedy solution, as many boxes as possible are fit into the truck. Therefore the number of boxes in greedy is equal to or more than the optimal solution's boxes.

Inductive Hypothesis: for the first k trucks, greedy has $n \geq m$ boxes.

Prove for truck k+1.

In the k+1 truck, assume the optimal can pack i boxes. Then greedy can also pack up to the ith box. Since greedy solution already had $n \geq m$ boxes, it can potentially pack more into the k+1 truck. Therefore the greedy solution will not switch trucks before the optimal solution since it could still potentially fit more into the k+1 truck. Thus the greedy fits into the minimum num of trucks.

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.

Solution:

Assume G has 2 MSTs, T and S.

This means that T and S have the same total cost.

Let e_1 be an edge with lowest weight in T but not S connecting v and w.

Then S must have a different edge e_2 connecting v and w.

If T and S have all the same edges otherwise, adding e_1 into S would create a cycle.

By definition of the problem, e_1 and e_2 have distinct weights and $c_{e_1} < c_{e_2}$.

Therefore, when creating a MST, e_1 would be chosen over e_2, but S has e_2, so S would not be an MST.

T therefore is the only MST for graph G and is thus distinct. ∎

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 $O(|E|)$.

Solution:

Let e=(v,w) be the new edge added
Use BFS to find the path in T between v and w
If e is most expensive edge then T is MST
else T is not MST

This algorithm works because the maximum edge in a cycle will never be in the MST.

BFS runs in O(|V|+|E|) but a minimum spanning tree has property |V|=|E|+1, so |V|<|E|. Therefore runtime becomes O(2|V|) = O(|V|) < O(|E|). The rest of the operations are constant time.

   (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 $O(|E|)$.

Solution:

Let e=(v,w) be the new edge added
Use BFS to find the cycle in T between v and w
Remove the highest value edge in the cycle

Similar to above, BFS runs in O(|V|+|E|) but |V|<|E| in an MST, so O(|V|) < O(|E|) and thus the runtime is 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.

Solution:

Cache size: 2

Requests: 1 2 3 1 2 1 3 1

Optimal # of Faults with FF: 5 (2 beginning, then 2, 3, 2)

# of Faults with FWF: 8 faults total

1, 2 → 2 faults
3, 1 → 2 faults
2, 1 → 2 faults
3, 1 → 2 faults

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

Solution:

Cache size: 3

Requests: 1, 2, 3, 4, 1, 2, 3, 1, 4

Optimal # of faults with FF: 5 (3 beginning, then 3, 2)

# of Faults with LRU: 8 faults total

1, 2, 3
4, 1, 2, 3
1, 2, 3
4, 2, 3, 4
3 beginning, then 5

[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.

5. Coding problem

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

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.

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
```