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: Ki Min Kang Wisc id: 908 404 5062

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:

Base case:
When there is only one truck, the solution is trivally optimal
Since we can only pack what fits into that one truck.
The base case holds

Induction Step:
Assume that for K trucks, First Fit (FF) has shipped at least as many boxes as any other strategy 'S'.
Now, we will show the statement also holds for K+1 trucks.
For strategy 'S' to ship more boxes than FF with the 'K+1'th truck, it would have to ship all the boxes FF shipped in its 'K+1'th truck plus at least one additional item 'm'. However, if FF did not ship item 'm' in its 'K+1'th truck, it implies that item 'm' could not fit in the truck because it would have exceeded the weight limit.

Therefore, the statement is true by induction.

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:

Let G has two different minimum spanning trees (MSTs), named T_1 and T_2. Then, we find an edge 'e' that is in T_1 but not in T_2. Since T_1 and T_2 are different, there must be such an edge. Next, we add the edge 'e' to T_2. Since T_2 is already an MST, adding any edge will make it a cycle and we will call this cycle 'C'. Within the cycle 'C', there must be an edge with the maximum cost, which we will call 'f'.

There are two scenarios that contradicts edge 'f':

1) If 'f' is the same as 'e', then removing 'f' from T_2 would still leave a spanning tree, which would be lighter than T_2 because we assumed that edge weights are distinct. This contradicts the assumption that T_2 is an MST.
2) If 'f' is not the same as 'e', then we can create a new tree T_3 by replacing 'f' with 'e' in T_2. T_3 has a lower cost than T_2, which again contradicts the assumption that T_2 is an MST.

Since both scenarios lead to a contradiction, the original assumption that there are at least two different MSTs must be false.

Therefore, the graph G must has a unique minimum spanning tree.

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:

The Graph G is represented as an adjacency list, and the MST is marked within this structure. To check if the MST is still valid after the insertion of a new edge, we will use the algorithm that checks if the new edge has a greater cost than any of the edges in the identified path. If it does not, the MST is no longer minimum-cost because replacing the highest-cost edge in the cycle with the new edge would yield a spanning tree with a lower total cost. Therefore, the runtime is 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 $O(|E|)$.

Solution:

We will use an algorithm that starts from one vertex v of the new edge, performing a DFS on T until the other vertex w is reached, noting the path taken. Along with the new edge, this path will form a cycle C. Within this cycle, identify the most expensive edge f. The MST is then updated by removing edge f and including the new edge. the runtime of both the DFS and the process of finding the maximum cost edge in the v to w path is O(E), where |E| is the number of edges in the graph. Therefore, 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:

Let's say request sequence $\sigma = \{x, y, z, x\}$, Cache size = 2

① Using FWF:
1) The cache initially loads 'x' (Page fault 1)
2) 'y' is requested and loaded into the cache. (Page fault 2)
3) 'z' is requested, FWF flushes both 'x' and 'y' because the cache is full then loads 'z' (Page fault 3)
4) Finally 'x' is requested and loaded (Page fault 4)

② Using FF:
1) The cache loads 'x' (Page fault 1)
2) 'y' is requested and loaded (Page fault 2)
3) 'z' is requested. FF looks ahead and finds that 'x' will be needed next, So it evicts 'y', which isn't needed soon, and loads 'z' (Page fault 3)
4) 'x' is requested, which is already in the cache, so there's no page fault.

Therefore, the examples above show that FF is more optimal

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

Solution:

Let's say request sequence $\sigma = \{x, y, z, x\}$, Cache size = 2

① Using LRU:
1) 'x' is loaded into the cache. (Page fault 1)
2) 'y' is requested and loaded into the cache (Page fault 2)
3) 'z' is requested, LRU evicts 'x', which was the least recently used, and loads 'z'. (Page fault 3)
4) 'x' is requested again, LRU evicts 'y', and loads 'x'. (Page fault 4)

② Using FF:
1) The cache loads 'x' (Page fault 1)
2) 'y' is requested and loaded (Page fault 2)
3) 'z' is requested. FF looks ahead and finds that 'x' will be needed next, So it evicts 'y', which isn't needed soon, and loads 'z' (Page fault 3)
4) 'x' is requested, which is already in the cache, so there's no page fault.

Therefore, the examples above show that FF is more optimal

[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. Your solution should be no worse than $O(nk)$ time, though try to aim for $O(n \log k)$ (where n is the number of page requests and k is the size of cache).

   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 size of 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 20 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
```