CS 577 **Assignment 4 – More Greedy** Spring 2024

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: Mahir Khan Wisc id: mhkhan9

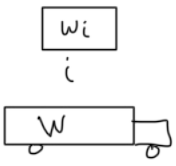
More Greedy Algorithms

> when $w_i \geq W$, $W = 0$, otherwise $W \mathrel{+}= w_i$

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.

Boxes



C = current weight -> if $C + w_i \geq W$, then T++ and C=0; otherwise, C += w_i

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: T = number of trucks returned by greedy algorithm

T^* = optimal number of trucks returned by an optimal solution

Assume that $|T| = k$, and $|T^*| = m$. We must show $m = k$.

We will prove the following statement: For all packages n from $n \leq k$, we have $|T| \leq |T^*|$ (i.e: the number of trucks required for $n \leq k$ packages in the greedy solution is always less than or equal to number of trucks in optimal sol.)

↳ Proof by induction:

For $n = 1$, the statement $|T| \leq |T^*|$ is obviously true. If $C + w_i > W$, then the max weight is reached, and both T and T^* will add 1 truck to number of trucks required. otherwise, the both stay at 0. Thus $|T| \leq |T^*|$ holds.

W = max possible weight

C = current weight in truck

w_i = weight of package i

Now, let $n > 1$. We will assume for our induction hypothesis that the statement is true for n-1, meaning $|T| \leq |T^*|$ holds for n-1 packages. In our inductive step, we will show the statement holds for n packages.

↳ Thus, for n packages, we must add 1 more package from the inductive hypothesis, and there are 2 scenarios:

↳ 1) $C + w_n > W$. In this scenario, there is no more space in the current truck for package n, thus there has to be another truck, especially since the order of packages cannot be changed. ∴ Both $|T| + 1$ and $|T^*| + 1$, ∴ our statement still holds. ∴ $|T| + 1 \leq |T^*| + 1$

↳ 2) $C + w_n \leq W$, meaning there is space in the truck for the nth package. ∴ $|T| \leq |T^*|$

We know that our greedy algo won't add another truck, and we can assume the optimal solution doesn't either, thus $|T| \leq |T^*|$ still hold! Thus by induction, we have shown that our greedy algo stays ahead or with an optimal solution.

Statement: the greedy algo returns the optimal # of trucks (minimized), $|T| = k$, $|T^*| = m$, (greedy, optimal)

↳ Proof by contradiction: If T is not optimal, that must mean $m < k$, meaning T has more trucks in the sol. than T^*. However, as seen above, we know that our greedy algo always keeps up / stays ahead, thus $m < k$ is a contradiction! ∎

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.
   ↳ No tiebreakers required.

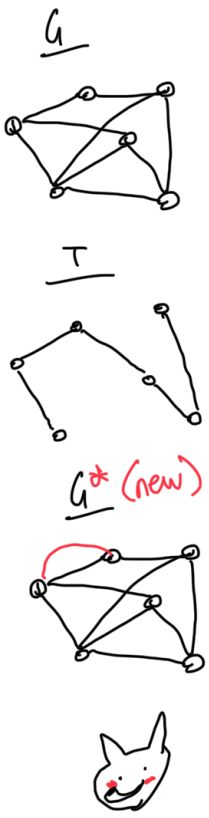
Solution: We can prove that G has a unique minimum spanning tree (MST) by first applying Kruskal's algorithm.

Kruskal's algorithm will build a spanning tree by successively inserting edges from G in order of increasing cost. Since we know that each edge has distinct edge costs, this means that Kruskal's algorithm does not have to deal with any tie breakers. Because of the fact that there's no tiebreakers, this means that there is a distinct order that Kruskal's algorithm creates the MST for graph G in. Thus due to the distinctness of the order, there must only be one unique MST for G.

To further prove this, we will conduct a proof by contradiction.

Assume that Kruskal's algorithm does NOT return a unique spanning tree, i.e: $|MST| > 1$. This means that there is some point during the execution of the algorithm where Kruskal's algo will have to make a choice between 2 edge costs. However, this is a contradiction because we know that G has distinct weights, thus the algo knows exactly what edge to evaluate at each iteration, thus no tie breakers needed.

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: * Check what nodes $v, w \in V$ that the new edge "e*" is connected to
* Compare the edge weight c_e between $v, w \in V$ in T to the new edge's edge weight c_{e*}
* If $c_e < c_{e*}$, then T remains the minimum cost tree for G, otherwise it is not.

```
Algorithm : optimalAfterNewEdge(T, e)
Input: T, an adjacency list representing the MST, and (e,c) the new edge in G with cost c | Output: True if still optimal, False if not optimal.
Let maxCost = 0
Let H = T //temporary tree
H = H+(e,c) //add new edge to temporary tree, causes a cycle
for each edge e ∈ H in H.
    if e.cost > maxCost:
        maxCost = e.cost
if c > maxCost:
    return True // the highest cost edge is cheaper than the new edge's edge cost, thus T is still optimal
else:
    return False
```

The traversal to find the max cost edge in T has runtime $O(|E|)$, for traversing all edges.
Thus the runtime is $O(|E|)$, since the if statements are constant.

(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:

```
Algorithm : updateMST(T, e)
Input: T, an adjacency list representing the MST, and (e,c) the new edge in G with cost c | Output: the new optimal MST with the new edge.
Let maxCost = 0
Let H = T //temporary tree
H = H+(e,c) //add new edge to temporary tree, causes a cycle
Let highestCostEdge = 0
for each edge e ∈ H in H.
    if e.cost > maxCost:
        maxCost = e.cost
        highestCostEdge = e
H.remove(highestCostEdge) //remove the highest cost edge in the tree
H.add(e) //add the new edge into the temp tree
return H // return new optimal MST
```

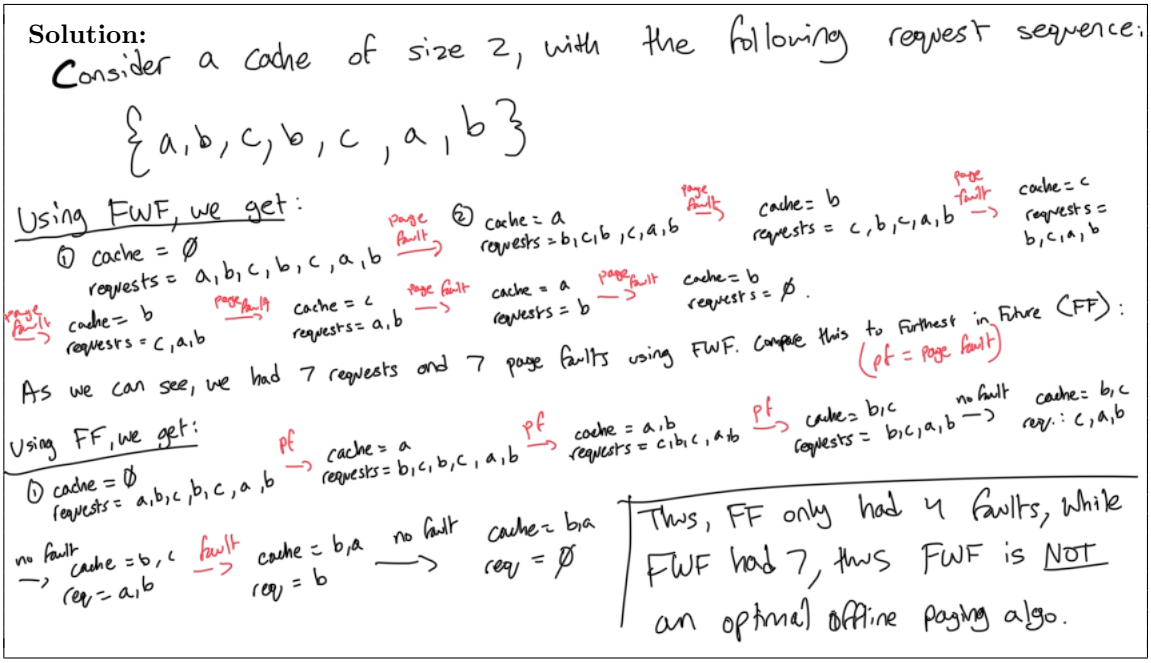
Iterating through every edge takes $O(|E|)$ time, everything else is constant
Thus total runtime is $O(|E|)$

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]

→ want to minimize the number of page faults.

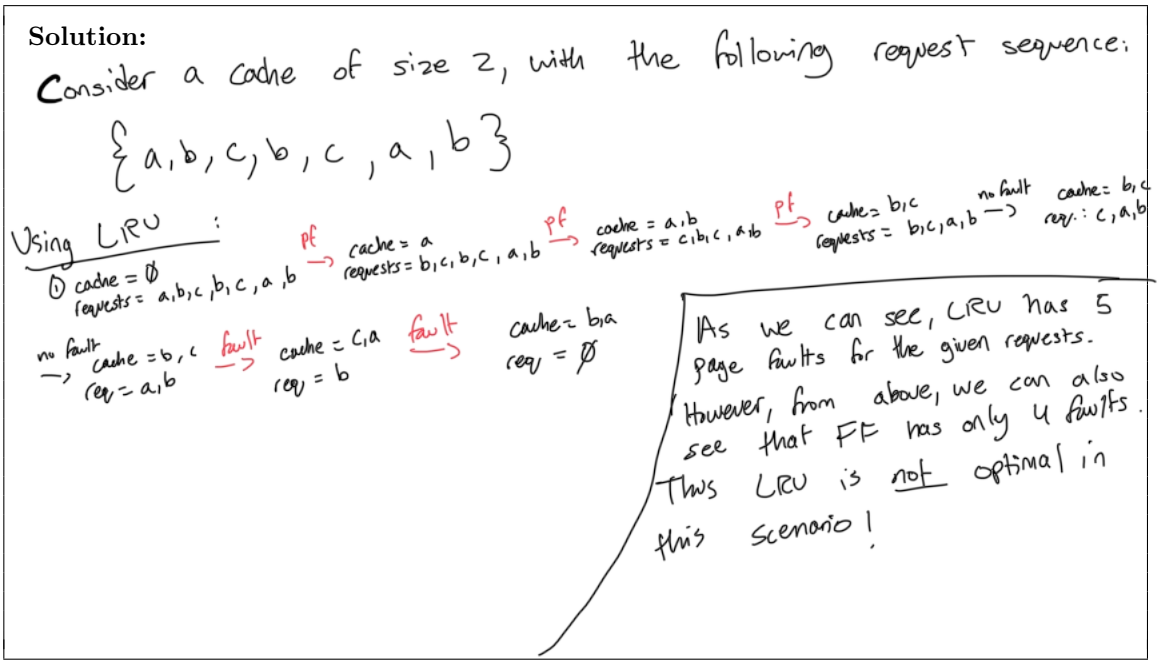
(a) FWF is a strategy that, on a page fault, if the cache is full, it evicts all the pages.

Solution:



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

Solution:



[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
```