# CS561, HW7

Ryan Scherbarth, University of New Mexico

October 2024

1. The count-min sketch discussed in class only provides count estimates of the number of times each item has been seen. What if we want to augment the sketch to also keep track of the $x$ items with largest count estimates for some value $x$? Describe how you would do this. You may find useful a heap; recall that a heap provides the functionality to (1) insert an item with a given value; (2) increase the key of an item already in the heap; and (3) delete min: delete the item in the heap with the minimum key. Assume these heap operations all take $O(\log(x))$ time. What is the run time of your augmented count-min sketch?

   We will create a min-heap that will store the most popular $x$ items.

   For every incoming item, we will start by estimating the items' count using the count-min, just as we did before.

   If the item is already in the heap we increase the count, if not we insert it as a new item in the heap with count 1. If we end up with a full heap then we will delete the minimum item before adding.

   To update the count-min we will take $O(k)$ time for the $k$ different hash functions we implement. Each of the hashing algorithms will take $O(\log(x))$ time, so the total running time will be $O(k + \log(x))$ where $k$ is the number of hash functions & $x$ is the number of items in the heap.

2. Consider the following alternative greedy algorithms for the activity selection problem discussed in class. For each algorithm, either prove or disprove that it constructs an optimal schedule

   - (a) Choose an activity with shortest duration, discard all conflicting activities and recurse

   This would not be an optimal solution, Lets say we have three schedule items;

   $i_1$ ranging from 1 to 4
   $i_2$ ranging from 3 to 5
   $i_3$ ranging from 5 to 8

   Our algorithm will always choose the shortest duration when possible, so we would first choose $i_2$ with a duration of 2, but then we have a conflict and cannot choose either $i_1$ or $i_2$.

   We can see this is not the ideal situation because it would be possible to choose $i_1$, $i_3$ for a total of 2 movies viewed.

- (b) Choose an activity that starts first, discard all conflicting activities and recurse.

Lets define a list of activities as follows;

$i_1$ ranging from 1 to 10
$i_2$ ranging from 2 to 3
$i_3$ ranging from 4 to 4
$i_4$ ranging from 6 to 9

Using this algorithm, we would just end up choosing $i_1$ and discard all of the remaining items.

We can clearly see this counterexample as the ideal situation here would be choosing every item besides $i_1$.

- Choose an activity that ends latest, discard all conflicting actives and recurse

Just as with the previous, we will slightly tweak our activities list;

$i_1$ ranging from 1 to 10
$i_2$ ranging from 2 to 3
$i_3$ ranging from 4 to 4
$i_4$ ranging from 6 to 9

Again in this example, we will end up choosing $i_1$ first, and disregarding all the others since they all conflict. We will again run into the issue of choosing the least optimal solution.

- (d) Choose an activity that conflicts with the fewest other activities, discard all conflicting activities and recurse.

$i_1$ ranging from 1 to 3
$i_2$ ranging from 1 to 3
$i_3$ ranging from 1 to 5
$i_4$ ranging from 4 to 7
$i_5$ ranging from 4 to 7
$i_6$ ranging from 6 to 10

We see in this case the algorithm will end up taking the greedy approach and choosing $i_1$, followed by $i_6$ due to it having the lead amount of conflicts. The flaw in this approach is when we have multiple duplicate values of the exact same time slot which could also be the ideal time slot. Having multiple duplicates means those end up having the most conflicts despite still being the optimal solution.

3. Now consider a weighted version of the activity selection problem. Imagine that each activity, $a_i$ has a weight, $w(a_i)$ (weights are totally unrelated to activity duration). Your goal is now to choose a set of non-conflicting activities that give you the largest possible sum of weights, given an array of start times, end times, and values as inputs.

   - (a) Prove that the greedy algorithm described in class - Choose the activity that ends first and recurse - does not always return an optimal schedule for this problem.

   The scheduling problem gets more difficult when we consider the weight of the total time behind each selection as well. Say we have the following;

   $i_1$ ranging from 1 to 2 (weight 2)
   $i_2$ ranging from 1 to 5 (weight 5)
   $i_3$ ranging from 5 to 6 (weight 2)
   $i_4$ ranging from 5 to 10 (weight 5)

   Here on the first iteration we choose $i_1$ because it ends first, and we disregard $i_2$ due to it's conflict. Next we choose $i_3$ because it ends sooner and again discard $i_4$ due to a conflict.

   We can clearly see this algorithm does not give us the ideal solution as we have finished with a total weight of 4, when choosing the alternative of $i_2$ and $i_4$ would result in a score of 10.

   - (b) Describe an algorithm to compute the optimal schedule in $O(n^2)$ time.
     Hint 1: Sort the actives by finish times. 2: let $m(j)$ be the maximum weight achievable from actives $a_1, a_2, \ldots, a_j$. 3: Come up with a recursive formulation for $m(j)$ and use dynamic programming. Hint 2: In the recursion in step 3, it'll help if you pre-compute for each job $j$, the value $x_j$ which is the largest index $i$ less than $j$ s.t. job $i$ is compatible with job $j$. Then when computing $m(j)$, consider that the optimal schedule could either include job $j$ or not include job $j$.

   We start by sorting the actives in increasing order of finish times, so the first item will be the one that finishes first.

   For each activity $j$, we pre-compute $x_j$ s.t. it represents the largest index $i$ where activity $a_i$ is compatible with $a_j$, meaning the activities do not overlap. We will set $x_j = 0$ if there is no possible $j$ to satisfy this requirement.

   Let $m(j)$ be the weight of the largest actives in the range $a_1, \ldots, a_j$.
   Base case: $m(0) = 0$
   Recursive case: $m(j) = \max(w(a_j) + m(x_j), m(j-1))$
   Where $w(a_j) + m(x_j)$ represents the case where we include a given activity, representing the weight of element $a_j$ and $m_j$ the maximum weight; and $m(j-1)$ is the case where we exclude an activity $j$.

   We can implement a dynamic programming table $DP$ and fill in the values of $m(j)$ for all $j$, which will take a total of $O(n^2)$ time. After this point, Computing $x_j$ for each activity will take $O(n)$, so the overall time complexity is $O(n^2)$.

4. Consider the following problem.
INPUT: Positive integers $r_1, r_2, \ldots, r_n$ and $c_1, c_2, \ldots, c_n$.
OUTPUT: An $n$ by $n$ matrix $A$ with 0/1 entries such that for all $i$ the sum of the $i$th row in $A$ is $r_i$ and the sum of the $i$th column in $A$ is $c_i$, if such matrix exists.
Think of the problem this way. You want to put pawns on a $n$ by $n$ chessboard so that the $i$th row has $r_i$ pawns and the $i$th column has $c_i$ pawns. Consider the following greedy algorithm that constructs $A$ row by row. Assume that the first $i-1$ rows have been constructed. Let $a_j$ be the number of 1's in the $j$th column in the first $i-1$ rows. Now the $r_i$ columns with maximum $c_j - a_j$ are assigned 1's in row $i$, and the rest of the columns are assigned 0's. that is, the columns that still need the most 1's are given 1's. Formally prove that this algorithm is correct using an exchange argument

Let $A$ be our primary matrix constructed using the above algorithm, and let $B$ be any other matrix that is valid according to the rules set above. $A$ and $B$ must differ in some positions, so we need to show that we can one by one substitute positions from $A$ into positions from $B$ and still have a valid solution at least as good as $A$.

We will exchange from $A$ to $B$ in 2x2 blocks. We can start by locating any 2x2 sub-matrix of $A$ s.t. two entries in row $i$ correspond to two entries in row $k$ of matrix $B$. When we find such a sub-matrix, we will swap the 2x2 squares.
Continue this algorithm until there are no longer any sub-matrices that satisfy the above condition. At each step after we make a swap we will verify that our solution matrix $A$ is still valid. In this case, we are replacing an algorithm that places the 1's in the rows and columns with the largest deficit with some column that still needs more 1's.

Based off of these results, we can conclude the swapping with matrix $B$, while still a valid algorithm, is one that is less valid than $A$. Since every possible alternative solution can be transformed back into solution $A$ while maintaining it's validity, we can finally conclude that $A$ is indeed the optimal solution.