CMSC 27200 - Problem Set 3

Sohini Banerjee

January 27, 2024

0

- Collaborators: Parnika Saxena, Kanchan Naik
- Sources: Lecture Notes, TA Office Hours

1

Algorithm

Overview: Sort the late fees in non-increasing order and visit the libraries in order of the sorted late fees. In other words, we greedily visit the library with the highest late fee, followed by the second highest late fee, and so on.

- Stored Data: Keep track of the current day x and total late fees *total*. Assume the late fees are given in a 1-based indexed array A, such that $A[i] = c_i$, where c_i denotes the late fee of library i for $1 \leq i \leq n$.
- Initialization: Sort A in decreasing order using the mergesort, such that $A[j] = c_{i_j}$, where c_{i_j} denotes the late fee of the library visited on day j. Let $x = 1$, starting at the library with the maximum late fee.
- Iterative Step: Add $(x-1) \cdot A[x]$ to *total* and increment x. If $x > n$, all libraries have been visited, so the algorithm terminates and returns *total*.

Proof

Definition. Let the greedy solution $S_{greedy} = (i_1, ..., i_n)$ where we visit library i_j on day j and pay a late fee of $(j-1) \cdot c_{i_j}$. Similarly, let the optimal solution $S^* = (i_1^*, ..., i_n^*)$ where we visit library i_j^* on day j^* and pay a late fee of $(j^*-1) \cdot c_{i_j^*}$. Furthermore, let $cost(S)$ denote the total late fees for a given solution S. Observe that if two libraries have the same late fee, they can be treated as copies of the same library.

Lemma. Let $S' = (i'_1, ..., i'_n)$ be a solution where we swap i_j^* and i_{j+1}^* if $c_{i_j}^* \leq c_{i_{j+1}}^*$. Since $cost(S^*) = \sum_{j^*=1}^{n} (j^*-1) \cdot c_{i_j}^*$ and $cost(S') = \sum_{j'=1}^{n} (j'-1) \cdot c'_{i_j}$,

$cost(S') \leq cost(S^*)$.

Since S_{greedy} visits the library in non-increasing order of late fees, we will show that swapping $c^*_{i_j} \leq c^*_{i_{j+1}}$ brings us closer to the greedy solution without reducing optimality (i.e. the cost does not increase).

Proof.

- $cost(S^*) - cost(S')$
- $= \sum_{j^*=1}^{n}(j^* - 1) \cdot c^*_{i_j} - \sum_{j'=1}^{n}(j' - 1) \cdot c'_{i_j}$
- $= (j^* - 1) \cdot c^*_{i_j} + j^* \cdot c^*_{i_{j+1}} + ((j' - 1) \cdot c'_{i_j} - j' \cdot c'_{i_{j+1}})$
  - Justification: Swapping i^*_j and i^*_{j+1} preserves the order of the other libraries visited, so late fees acquired from those libraries remain unchanged. Therefore, the difference $cost(S^*) - cost(S')$ is only affected by the late fees acquired from i^*_j and i^*_{j+1}, which when swapped are i'_{j+1} and i'_j, respectively.
- $= (j^* - 1) \cdot c^*_{i_j} + j^* \cdot c^*_{i_{j+1}} - ((j^* - 1) \cdot c^*_{i_{j+1}} + j^* \cdot c_{i^*_j})$
  - Justification: $c^*_{i_j} = c'_{i_{j+1}}$ and $c^*_{i_{j+1}} = c'_{i_j}$ because we swapped i^*_j and i^*_{j+1}, so the late fee for the libraries visited on those days switch.
- $= j^* \cdot c^*_{i_j} - c^*_{i_j} + j^* \cdot c^*_{i_{j+1}} - j^* \cdot c^*_{i_{j+1}} + c^*_{i_{j+1}} - j^* \cdot c^*_{i_j}$
- $= -c^*_{i_j} + c^*_{i_{j+1}}$

At this point, we have that $cost(S^*) - cost(S') = -c^*_{i_j} + c^*_{i_{j+1}}$. Since we assumed $c^*_{i_j} \leq c^*_{i_{j+1}}$, then $-c^*_{i_j} + c^*_{i_{j+1}} \geq 0$, so $cost(S^*) - cost(S') \geq 0$. This means that $cost(S') \leq cost(S^*)$, which is what we wanted to prove.

Therefore, for any optimal solution S^*, we can switch the order of libraries visited if $c^*_{i_j} \leq c^*_{i_{j+1}}$, producing a solution closer to S_{greedy} without reducing total late fees since $cost(S') \leq cost(S^*)$. This proves that S_{greedy} is a correct solution because it visits each library and acquires the minimum total late fees.

Runtime

First, we use mergesort to sort array A, which takes $O(n \log n)$ time. Then, we iterate through each element in A, which takes $O(n)$ time. For each element, we add to the total late fees, which takes $O(1)$ time. Thus, the time complexity of our algorithm is $O(n \log n)$, which is polynomial time.

2a

Algorithm

Overview: Use the maximum number of \$10 bills possible, followed by the maximum number of \$5, \$2, and \$1 bills until we reach N dollars. Intuitively, we greedily use as many large bills as possible before using smaller bills to reduce the total number of bills used.

- Stored Data: Keep track of the number of \$10 bills used c_{10}, \$5 bills used c_5, \$2 bills used c_2, and \$1 bills used c_1. Furthermore, keep track of the remaining dollars we need to pay r.
- Initialization: Let $c_{10} = c_5 = c_2 = c_1 = 0$ as no bills have been used yet. Let $r = N$ because we have not paid any dollars yet.
- Iterative Step:
  - While $r \geq 10$, increment c_{10} by 1 and decrement r by 10.
  - While $r \geq 5$, increment c_5 by 1 and decrement r by 5.
  - While $r \geq 2$, increment c_2 by 1 and decrement r by 2.
  - While $r \geq 1$, increment c_1 by 1 and decrement r by 1.
  - Return c_{10}, c_5, c_2, and c_1, denoting the number of each bill type used.

2b

Proof.

Lemma 1. The greedy algorithm produces a legitimate solution.

Proof. We will show that at the very least, once we cannot use \$10, \$5, or \$2, we can use \$1 bills to pay N. We keep adding \$10 bills until the remaining dollars fall below \$10. Then, we keep adding \$5 bills until the remaining dollars fall below \$5. Then, we keep adding \$2 bills until the remaining dollars fall below \$2. In this case, we either have \$1 or \$0 left to pay. If the former, we use a \$1 bill and if the latter, we are done. Thus, the greedy algorithm always produces a legitimate solution.

Lemma 2. The greedy algorithm is the optimal solution using \$5, \$2, and \$1 bills.

Proof. Let c_5 be the number of \$5 dollar bills used, c_2 be the number of \$2 bills used, and c_1 be the number of \$1 bills used. Let the greedy solution $S_{greedy} = (c_1, c_2, c_5)$ and the optimal solution $S^* = (c'_1, c'_2, c'_5)$. We can use an exchange argument to show that S_{greedy} is the optimal solution. First, note that $5c'_1 + 2c'_2 + c'_1 = 5c_5 + 2c_2 + c_1 = N$.

- Suppose $c'_5 = c_5$. This means $2c'_2 + c'_1 = 2c_2 + c_1$.
  - Suppose $c'_2 = c_2$. This means $c'_1 = c_1$. In this case, $S^* = S_{greedy}$.
  - Suppose $c'_2 < c_2$. This means $c'_1 > c_1$ because otherwise, $2c'_2 + c'_1 < 2c_2 + c_1$. In this case, S^* has to make up for at least \$2 with \$1 bills. Let x be the amount of of money we have to make up for. The only case is exchange $x = 1 + 1$ with 2, which is what S_{greedy} does. For $x > 2$, we should maximize the number of \$2 bills used because otherwise we have to compensate for \$2 with \$1 bills and end up with the $1 + 1$ case above.
- Suppose $c'_5 < c_5$. This means $2c'_2 + c'_1 > 2c_2 + c_1$ because otherwise $5c'_1 + 2c'_2 + c'_1 < 5c_5 + 2c_2 + c_1$.
  - In this case, S^* has to make up for at least \$5 with \$2 and \$1 bills. Let x be the amount of money we have to make up for. The first case is exchange $x = 2+2+1$ with 5 and the second case is exchange $x = 2+2+2$ with $5+1$, both of which S_{greedy} does. As shown above, no solution can contain $x = 1 + 1$, so these are the combinations we change with a 5. For $x > 6$, any combination of \$2 and \$1 always contains at least 3 \$2 bills (since assuming did prior $1 + 1$ exchange so have maximum 1 \$1 bill). We can substitute this for 5 (if a \$1 is being used) or $5 + 1$ (if a \$1 bill is not being used). For all x, we cannot use $2 + 2 + 1$, $2 + 2 + 2$, or $1 + 1$, meaning we should exchange these combinations with \$5 or \$2 bills until none of these combinations are present, which is what S_{greedy} does.

Lemma 3. The greedy algorithm is the optimal solution using \$10, \$5, \$2, and \$1 bills.

Proof. Let c_{10} be the number of \$10 bills used, c_5 be the number of \$5 dollar bills used, c_2 be the number of \$2 bills used, and c_1 be the number of \$1 bills used. Let the greedy solution $S_{greedy} = (c_1, c_2, c_5, c_{10})$ and the optimal solution $S^* = (c'_1, c'_2, c'_5, c'_{10})$. We can use an exchange argument to show that S_{greedy} is the optimal solution. First, note that $10c'_{10} + 5c'_1 + 2c'_2 + c'_1 = 10c_{10} + 5c_5 + 2c_2 + c_1 = N$.

- Suppose $c'_{10} = c_{10}$. Then, $c'_1 + 2c'_2 + 5c'_5 = c_1 + 2c_2 + 5c_5$. The problem reduces to using \$5, \$2, and \$1, which we proved in Lemma 2.
- Suppose $c'_{10} < c_{10}$. This means $c'_1 + 2c'_2 + 5c'_5 > c_1 + 2c_2 + 5c_5$ because otherwise $10c'_{10} + 5c'_1 + 2c'_2 + c'_1 < 10c_{10} + 5c_5 + 2c_2 + c_1$.
  - In this case, S^* has to make up for at least \$10 with \$5, \$2, and \$1 bills. Let x be the amount of money we have to make up for. The only case is exchange $x = 5 + 5$ for 10, which is what S_{greedy} does. For $x \geq 10$, we must use at least two \$5 bills, because otherwise, we would have to makeup for at least \$5 with \$2 and \$1 bills, which

Lemma 1 showed can be exchanged. Therefore, as long as $x \geq 10$, we can exchange $5 + 5$ with 10, and eventually this makes $x < 10$, which reduces to the Lemma 2 problem of using only \$5, \$2, and \$1 bills, which is what S_{greedy} does.

Thus, we have shown that the greedy solution is optimal. Furthermore, the solution is unique because any deviation from the greedy solution can be optimized with the exchange argument, typically by replacing lower denomination bills with a higher denomination bill. The greedy solution always gives a unique answer, and the exchange argument shows that any other solution can be exchanged to become more similar to the greedy solution and eventually exactly the same. This means that the greedy solution is optimal and unique.

2c

No, the algorithm will not always work. A counterexample is $N = 10$. The greedy algorithm takes the following steps:

- Let $c_8 = c_5 = c_1 = 0$ and $r = 10$.
- $r = 10 \geq 8$, so $c_8 = 1$ and $r = 2$.
- $r = 2 < 5$.
- $r = 2 \geq 1$, so $c_1 = 1$ and $r = 1$.
- $r = 1 \geq 1$, so $c_1 = 2$ and $r = 0$.
- $r - 0 < 1$.

Thus, the greedy algorithm selects $c_8 = 1, c_5 = 0, c_1 = 2$, for a total of 3 bills. This is not optimal because we can achieve \$10 with $c_8 = 0, c_5 = 2, c_1 = 0$, for a total of 2 bills.

3a

Overview: We will use induction and a proof by contradiction to show that every blue tree is indeed a tree. Note that we already know each blue tree is connected (Ed 213), so we simply have to prove that it is not a cycle and therefore a tree.

Inductive Hypothesis: We will induct over the phases i. The base case $i = 1$ is trivial because every vertex is disconnected by definition, and single vertex graphs are always trees. The inductive hypothesis is that every blue tree is a tree in phase $i - 1$. We will show that this implies every blue tree is a tree in phase i.

Assume for contradiction that the algorithm produces a cycle C in phase i. Furthermore, let $T = \{T_1, ..., T_k\}$ be the components produced in phase $i - 1$ that form cycle C in phase i. With the inductive hypothesis, we know that $t \epsilon T$ is a tree. By definition, cycle C must have more than $k - 1$ edges. In addition, each component of cycle C must have more than one edge because by definition, every edge can be removed and there should still exist a path to the remaining components (Ed 211).

Now, given that all edge weights are distinct, let e_{max} be the edge of maximum weight added in phase i to cycle C. We know that e_{max} connects some component T_i to another component T_j. However, since each component has more than one edge, both T_i and T_j have a smaller and minimum edge e_i and e_j, respectively, such that $e_i < e_{max}$ and $e_j < e_{max}$. In this case, since each component $t \epsilon T$ finds the smallest edge with a single endpoint in t that connects it to some other component, T_i would select e_i and T_j would select e_j over the higher weighted edge, e_{max}. Therefore, neither T_i nor T_j would select e_{max}, which is a contradiction to assuming it is the largest weighted edge in cycle C in phase i.

Therefore, the inductive hypothesis holding for phase $i - 1$ shows that it also holds for phase i. So, for all phases $i \geq 1$, every blue tree is indeed a tree.

3b

Overview: Let us call the tree our algorithm produces T and the true minimum spanning tree MST. From part a, we know that T is indeed a tree such that all vertices of G are in T. Now, we must show that T is a minimum spanning tree. Note that there is a unique minimum spanning tree for G since all edge weights are distinct. We will use a direct proof to show that every edge selected by our algorithm is in the minimum spanning tree.

Theorem. Each edge $e = (u, v) \epsilon E$ is in MST if and only if there exists a cut

(L, R) such that e is the shortest edge between L and R.

Proof. Lecture notes.

Lemma. The resulting tree T is a spanning tree.

Proof. Suppose we are on some phase i. Let L denote some component produced in phase $i-1$ and R denote $V - L$, where V is all the vertices of the graph G. We can make the following notes:

- L and R are not connected because otherwise they would belong to the same component, and R could not be represented as $V - L$.
- The vertices in R may not necessarily be connected in phase i, but they will be connected by the algorithm's termination because the algorithm terminates when all vertices belong to the same component, as the algorithm produces a spanning tree, which we proved in part a.

By the algorithm, we will select an edge that has one endpoint in L with the smallest weight, connecting it to another tree. We can denote this edge e'. By note 2, we know that since all vertices will be connected, edge e' can be used to separate vertices in L from vertices in R. Therefore, edge e' produces a cut (L, R) such that e' is the minimum weight edge connecting L and R. By the Theorem, this means e' is in MST.

As a result, every edge selected by the algorithm is an edge is in the MST. Furthermore, since both T and MST are spanning trees of graph G, they both contain n vertices and $n-1$ edges. So, since every edge by the algorithm is in MST, T and MST but have exactly the same edges and thus, are the same tree. This proves that the resulting tree of the algorithm is a minimum spanning tree.

3c

Suppose there are n nodes, numbered x_1 to x_n. Also note that we count initialization as a phase.

The **minimum** number of phases is 2. We can define the graph as follows.

- We initialize our current edge weight $w = 1$. For each $i \epsilon [1, n-1]$, we add an edge $(i, i+1)$ of weight w, then increment w before adding the next edge.
- For example, if $n = 4$, the edges produced are (in the form of (u, v, w)):
  - $(1, 2, 1), (2, 3, 2), (3, 4, 3)$.
- If we simulate the algorithm, we get the following phases:

  - $\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}$
  - $\{x_1, x_2, x_3, x_4\}$ after adding edge $(1, 2, 1)$ for x_1 and x_2, $(2, 3, 2)$ for x_3, and $(3, 4, 3)$ for x_4
- As shown, this takes the minimum of 2 phases.

This works because the latter $n-1$ vertices all select a different edge, produces $n-1$ new edges. Also, x_1 selects the same edge as another vertex, so in 2 phases, we end up with $n-1$ edges, forming the tree. This answer is tight because we have an example that demonstrates this and know there cannot be 1 phase for a general n since then the graph is not connected.

The **maximum** number of phases is $floor(\log_2 n) + 1$. We can define the graph as follows.

- We initialize our current edge weight $w = 1$ and a queue with the current components as lists: $[[x_1], [x_2], ..., [x_{n-1}], [x_n]]$
- While $len(queue) > 1$, we pop the first two elements of the queue, A and B. We add an edge of the current weight between the last vertex of A and first vertex of B. Then we increment the weight and add the list concatenation $A + B$ to the back of the queue.
- For example, if $n = 7$, the edges produced are (in the form of (u, v, w)):
  - $(1, 2, 1), (3, 4, 2), (5, 6, 3), (7, 1, 4), (4, 5, 5), (2, 3, 6)$
- If we simulate the algorithm, we get the following phases:
  - $\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}, \{x_7\}$
  - $\{x_1, x_2, x_7\}, \{x_3, x_4\}, \{x_5, x_6\}$ after adding edge $(1, 2, 1)$ for x_1 and x_2, $(3, 4, 2)$ for x_3 and x_4, $(5, 6, 3)$ for x_5 and x_6, and $(7, 1, 4)$ for x_7
  - $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$ after adding edge $(4, 5, 5)$ for $\{x_3, x_4\}$ and $\{x_5, x_6\}$, and $(2, 3, 6)$ for $\{x_1, x_2, x_7\}$ and $\{x_3, x_4\}$
- As shown, this takes the maximum of $floor(\log_2 7)+1 = floor(2.81)+1 = 2 + 1 = 3$ phases.

This works because at each phase, we must reduce the number of components by at least half. If we have an even number of components, we simply pair components with each other, reducing the components by exactly half. If we have an odd number of components, we pair components with each other until there is one component left, which joins a pair of components, reducing the components by more than half. Therefore, it takes a maximum of $floor(\log_2 n)+1$ phases (adding 1 for initialization). This answer is tight because we have an example that demonstrates this and know the phases cannot exceed it.

4

Algorithm

Overview: If the value at the current index is not a duplicate, compare the number of spaces to fill to the right of this index and the number of distinct numbers we have remaining. This determines whether to search in the left or right subarray for the duplicate.

- Stored Data: We are given a sorted array A. Keep track of the $left$ and $right$ indices of the subarray of A.
- Initialization: Initialize $left = 0$ and $right = n - 1$, since our subarray is currently A.
- Iterative Step: We find the midpoint index mid of the subarray. If the value at mid is a duplicate, we return the index mid or $mid-1$ (depending on whether the duplicate element occurs to the left or right of mid). We determine the number spaces to fill in the right subarray and the number of elements that can lie to the right of mid.
  - number of spaces to fill $= (n-1) - mid$: this determines the number of index positions to the right of mid
  - number of elements that can lie to the right of $mid = (n-1) - A[mid]$: since $A[mid]$ is not a duplicate, the elements to the right of mid must be strictly higher than $A[mid]$ and at most $n - 1$
- If there are more spaces to fill than the number of elements that can fill them, there must be a duplicate in the right subarray, so we set $left = mid + 1$. Otherwise, there must be a duplicate in the left subarray, so we set $right = mid - 1$.

```
set left = 0, right = n - 1
while left < right, do:
    set mid = floor((left + right) / 2)

    if mid - 1 in bounds and A[mid] == A[mid - 1]: return mid
    if mid + 1 in bounds and A[mid] == A[mid + 1]: return mid + 1

    set spaces_to_fill = (n - 1) - mid
    set numbers_of_elements = (n - 1) - A[mid]

    if spaces_to_fill > numbers_of_elements, do:
        set left = mid + 1
    else, do:
        set right = mid - 1
```

Explanation

We make use of the pigeonhole principle in this problem and the bounds of elements allowed to appear in A. If we consider the midpoint element and it is a duplicate, we have solved the problem. Otherwise, the duplicate (which we know is present) occurs either between $mid + 1$ and $right$ or $mid - 1$ and $left$.

- Consider the number of spaces to fill to the right. This is the difference between the maximum index and current index.
- Then, consider the number of elements we can fill these spaces with. Our maximum element is $n - 1$, and the minimum element is greater than the element at midpoint. Therefore, the number of possible elements to the right is the difference between the maximum element and the current element.

From, here we have two cases:

- The number of spaces exceeds the number of possible elements. In this case, by the pigeonhole principle, it is clear that a duplicate must occur in the right subarray.
- The number of spaces does not exceed the number of possible elements. In this case, it is possible the right subarray contains a duplicate or it does not. However, due to the constraint of numbers being from 1 to $n - 1$, overall there are more spaces to fill than numbers available. Since the right subarray has enough numbers to fill the spaces, the left array must not. Therefore, regardless of whether a duplicate occurs in the right subarray, we are guaranteed a duplicate in the left subarray, and reduce our search to that.

By narrowing down the subarray each time, we will find the element that is identical to at least one of its neighbors, thus finding the duplicate.

Runtime

Using a binary search algorithm, we narrow our search array length by half, either narrowing to $left$ and $mid - 1$ or to $mid + 1$ and $right$. This means that the runtime is $O(\log n)$. Note that the runtime does not include sorting the A since it is already sorted.

We can also consider the recurrence relation. The recursive step is called $\log n$ times and within each step, it takes $O(1)$ time to check whether a duplicate exists at mid and update the bounds. Therefore, the recurrence relation is $T(n) = T(\frac{n}{2}) + 1$, which has a time complexity $O(\log n)$ as showed in the lecture.