CSCI 670: Advanced Analysis of Algorithms
3rd Assignment

Fall 2021

1 Homework Problems

Problem 1. Let $a_i \in \mathbb{R}$ be a sequence of reals. A sequence is said to be unimodal if there exists a k such that

- $a_{i-1} < a_i$ for all $i \leq k$
- $a_i > a_{i+1}$ for all $i \geq k$.

Design an $O(\log n)$ algorithm to find the index k which is also the maximum value element. Similarly, derive an algorithm to find the maximum value of a unimodal function $f(x) : \mathbb{R} \to \mathbb{R}$ of reals up to some $\varepsilon > 0$ precision? What will be the runtime of that algorithm?

Solution 1. We will implement an algorithm similar to binary search. However its decision mechanism will be different. The following algorithm returns the index of the maximum element.

Algorithm 1 Search(a, begin,end)

```
if begin=end then
    Output: begin
end if
Let m_1 = begin + (begin + end)/3.
Let m_2 = end − (begin + end)/3.
if a_{m_1} ≤ a_{m_2} then
    Output: Search(a, m_1, end)
else
    Output: Search(a, begin, m_2)
end if
```

First, we show that algorithm outputs the correct index always. It is clearly true for the base case begin = end. We apply induction on the search array lenght. Assume that algorithm outputs the correct output for any unimodal array with length strictly less than k, and begin $-$ end $= k$. We will investigate two cases.

- $a_{m_1} \leq a_{m_2}$. Observe that maximum element can not be in the interval of begin : m_1 since the array is unimodal. Thus, it is enough to search between a_{m_1} and end. Notice that $a_{m_1}, \ldots a_{\text{end}}$ is also a unimodal array.
- $a_{m_1} > a_{m_2}$. This case is symmetric to the previous one.

Thus, in each iteration we branch with the correct subarray to search the maximum element and the size of the problem is decreases. By induction, the algorithm finds the correct index.

Since we shrink the problem size by 2/3 in each branching, it takes $\log_{\frac{3}{2}} n$ steps to find a singleton. Thus, the runtime of the algorithm is $O(\log n)$. In this discussion we omit the details about indices being integer, one can keep track of rounding m_1 and m_2 to nearest integer.

Problem 2. Given n points in the plane, describe an algorithm to find the(a) closest pair in $n\log(n)$ time.

Solution 2. Read the Section 5.4 of the textbook [2].

Problem 3. Consider n distinct points in the 2D plane, $(x_1, y_1), \ldots, (x_n, y_n)$. We say that point i dominates point j if $x_i \geq x_j$ and $y_i \geq y_j$. Design an $O(n \log n)$ divide and conquer algorithm that finds all points that are not dominated by any other point.

Solution 3. We can modify the merge sort algorithm (sorting points by their x coordinate first and then by the y coordinate) slightly to also mark the points that are dominated by another point. The only modification is that after merging the sorted array on the left with the sorted array on the right, we do one more step on the merged array to mark all of its points that are dominated by another point of it. Assume that after merging we have a sequence of points $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$. As the sequence is sorted, we have that $x_1 \leq x_2 \leq \cdots \leq x_k$ and for all $j < i$ such that $x_j = x_i$ we have $y_j < y_i$. Because of this we have that if (x_i, y_i) dominates (x_j, y_j) then $i > j$. To find all points that are dominated, we can scan the array from right to left keeping the maximum value of y observed so far. At point j, if the maximum value of y observed so far (excluding y_j) is at least y_j, then we mark the point j as dominated. The complexity of the algorithm is $\Theta(n \log n)$ as it is given by the recurrence $T(n) = 2T(n/2)+\Theta(n)$.

Problem 4. Given real numbers $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, design an algorithm to solve the following equation in linear time (assuming that each simple math operation takes $O(1)$ time).

$$\begin{bmatrix} a_1 & -1 & 0 & 0 & 0 & \ldots \\ -1 & a_2 & -1 & 0 & 0 & \ldots \\ 0 & -1 & a_3 & -1 & 0 & \ldots \\ & \ddots & \ddots & \ddots & \ddots & \\ \ldots & 0 & 0 & -1 & a_{n-1} & -1 \\ \ldots & 0 & 0 & 0 & -1 & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

Solution 4. There exists a constant-size separator such that partitions the matrix into two matrices of size at most $\frac{n}{2}$. You can recursively keep doing this. Thus, the running time of this algorithm is $T(n) = 2T(n/2) + O(1)$ which is linear in n.

Problem 5. Chapter 5, exercise 3 of the textbook [2]. Can you think of an improved algorithm to solve the decision problem that makes only $O(n)$ invocations to the equivalence tester?

Solution 5. A set of cards is defined to contain a majority account if more than half its cards belong to a single account. Pair the cards arbitrarily into $\lfloor \frac{n}{2} \rfloor$ pairs. If n is odd, one card is left unpaired.

In each pair, if the cards are not equivalent, throw both out. If they are equivalent, throw one out.

We have thus reduced the number of cards by at least $\lfloor \frac{n}{2} \rfloor$, since in each pair at least one card gets thrown out. Recursively perform this until we are left with either one or zero cards.

The remaining set of cards contains a majority account if the original set contained a majority account. Hence if the original set contains a majority element, the above procedure finds it (one card remains in the end). If the original set does not contain a majority element, the procedure might still terminate in a single card. So as a final step in our algorithm, if

one card remains, we compare it with the every other card to verify if it is indeed a majority element.

If $T(n)$ denotes the running time of the corresponding divide-conquer algorithm,

$$T(n) \leq T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n) \Rightarrow T(n) = \mathcal{O}(n)$$

Problem 6. Chapter 5, exercise 5 of the textbook [2]

Solution 6. We can design $O(n \log n)$ algorithm. First of all, sort lines decreasing slopes of a_i. We will apply a divide&conquer algorithm. We first split set of lines into two sets from the middle and find visible surfaces of each set separetely. Notice that the problem size is divided by two in each level.

Once we obtain the solutions of each sides, we can iterate elements simultaneously and remove the hidden surface starting from middle (rightmost line of left side and leftmost line of right side). In each step we will determine whether any of the current lines are blocked by the other side by finding intersection points. This step can be implemented in linear time. Thus, run time of the algorith is $T(n) \leq 2T(n/2) + O(n) = O(n \log n)$.

Problem 7. Chapter 5, exercise 6 of the textbook [2]

Solution 7. Let r be the median of the tree T, we call r as root of the tree T. Observe that in each step we go down from the root, the size of the subtree divides by 2. See the following algorithm to find the local minimum.

Algorithm 2 LocalMin(x, T)

Let r be the root of tree T.
if r is a local minimum or $|T| = 1$ **then**
  Output r
else
  Let v be the neighbor of r such that $x_v < x_r$.
  Output LocalMin(x, T_v) where T_v is the v rooted subtree of T.
end if

One can show that the algorithm always outputs a local minimum (How?, remember the median of tree algorithm). As we discussed in the previous paragraph, the size of the tree divides by two in each recursive step. Thus, it takes $O(\log n)$ iteration to find a local minimum. Since we make constant number of probing operations in each step, the algorithm probes $O(\log n)$ vertices.

Problem 8. Given a vector $\mathbf{x} = (x_1, \dots x_n)$, the circulant matrix induced by $\mathbf{x}$ is the matrix

$$C = \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_{n-1} & x_n \\ x_n & x_1 & x_2 & \dots & x_{n-2} & x_{n-1} \\ x_{n-1} & x_n & x_1 & \dots & x_{n-3} & x_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ x_3 & x_4 & x_5 & \dots x_1 & x_2 & \\ x_2 & x_3 & x_4 & \dots x_n & x_1 & \end{bmatrix}$$

(a) Let F_n be the Fourier matrix of order n. That is,

$$F_n = \begin{bmatrix} 1 & 1 & \dots 1 & 1 \\ 1 & \omega & \dots \omega^{n-2} & \omega^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-2} & \dots \omega^{(n-2)^2} & \omega^{(n-2)(n-1)} \\ 1 & \omega^{n-1} & \dots \omega^{(n-1)(n-2)} & \omega^{(n-1)^2} \end{bmatrix}$$

where ω is the n-th root of unity. Verify that $C = F_n \cdot \mathrm{diag}\,(F_n\mathbf{x}) \cdot F_n^{-1}$ (updated). (b) Explain why for any vector $\mathbf{v}, C\mathbf{v}$ can be computed in $O(n \log n)$ time.

Solution 8. (a) The verification relies on the fact that $\sum_{i=0}^{n-1} \omega^i = 0$.

Consider F_n as a function of ω, one can check that $F_n^{-1} = \frac{1}{n} F_n(\omega^{-1})$ (inverse fourier transform). Next, we define the polynomial $p_x(t) = \sum_{i=1}^{n} x_i \cdot t^{i-1}$. Observe that $F_n\mathbf{x} = (p_x(1), p_x(\omega), \ldots p_x(\omega^{n-1}))^T$. Now, we will compute the each entry $C_{i,j}$.

$$
\begin{aligned}
C_{i,j} &= \sum_{k=1}^{n} [F_n]_{i,k} \cdot [diag(F_n\mathbf{x}) \cdot F_n^{-1}]_{k,j} \\
&= \sum_{k=1}^{n} [F_n]_{i,k} \cdot \left(\sum_{\ell=1}^{n} diag(F_n\mathbf{x})_{k,\ell} \cdot [F_n^{-1}]_{\ell,j} \right) \\
&= \sum_{k=1}^{n} [F_n]_{i,k} \cdot \left(diag(F_n\mathbf{x})_{k,k} \cdot [F_n^{-1}]_{k,j} \right) && \text{(only diag. is non-zero)} \\
&= \sum_{k=1}^{n} \frac{1}{n} \omega^{(i-1)\cdot(k-1)} \cdot p_x(\omega^{k-1}) \cdot \omega^{-(k-1)\cdot(j-1)} \\
&= \sum_{k=1}^{n} \frac{1}{n} \omega^{(k-1)\cdot(i-j)} \cdot \sum_{\ell=1}^{n} x_\ell \cdot \omega^{(k-1)\cdot(\ell-1)} \\
&= \sum_{\ell=1}^{n} x_\ell \cdot \frac{1}{n} \cdot \sum_{k=1}^{n} \omega^{(k-1)\cdot(i-j+\ell-1)} && (\sum_{i=0}^{n-1} \omega^i = 0) \\
&= x_{(j-i \mod n)}
\end{aligned}
$$

(b) By part (a), the product $C\mathbf{v}$ is equal to $F_n^{-1}\,\mathrm{diag}\,(F_n\mathbf{x})\,F_n \cdot \mathbf{v}$, which can be computed in the following way:

i. Compute $F_n\mathbf{x}$ using FFT (i.e. divide and conquer), which takes $O(n \log n)$ time;
ii. Compute $F_n^{-1}\mathbf{v}$ using FFT, which takes $O(n \log n)$ time;
iii. Compute the pointwise product $\mathbf{u} = \mathrm{diag}\,(F_n\mathbf{x}) \cdot \left(F_n^{-1}\mathbf{v}\right)$, which takes $O(n)$ time;
iv. Compute $F_n\mathbf{u}$ using inverse FFT (also divide and conquer), which takes $O(n \log n)$ time.

Problem 9 (Problem from 2nd homework)**.** Consider two subsets of $\{1, 2, \ldots, n\}$: A and B. The direct sum of A and B is defined as $A + B = \{a + b : a \in A, b \in B\}$. Design an $O(n \log n)$ algorithm to find the direct sum.

Solution 9. Let us represent A and B as binary arrays a_i and $b_i (i = 1..n)$ correspondingly, with $a_i = 1$ iff $i \in A$ and $b_i = 1$ iff $i \in B$. Consider the polynomials $a(x) = a_1x + a_2x^2 + \cdots + a_nx^n$ and $b(x) = b_1x + b_2x^2 + \cdots + b_nx^n$. Let $c(x) = c_1x + c_2x^2 + \cdots + c_{2n}x^{2n}$ be the product of $a(x)$ and $b(x)$. It is easy to see that $c_i \geq 1$ if and only if $i \in A + B$. Therefore, to find $A + B$ we just need to find the nonzero coefficients of $c(x)$. Computing the coefficients of $c(x)$ can be done in $O(n \log n)$ time using the discrete FFT algorithm.

Problem 10. .

(a) [1D correlation] Consider two sequences of numbers: $a_0, a_1, \ldots, a_{n-1}$ and $b_0, b_1, \ldots, b_{m-1}$, where $n \geq m$. The 1D discrete correlation of a and b is a sequence c of size $n - m + 1$ such that:

$$c_k = \sum_{i=0}^{m-1} a_{k+i} b_i, \quad k = 0, \ldots, n - m$$

Design an $O(n \log n)$ algorithm for computing 1D discrete correlation.

(b) [2D correlation] Now consider two matrices: $A \in \mathbb{R}^{n_1 \times n_2}$ and $B \in \mathbb{R}^{m_1 \times m_2}$, such that $m_1 \leq n_1$ and $m_2 \leq n_2$. The 2D discrete correlation of A and B is a matrix of size $(n_1 - m_1 + 1) \times (n_2 - m_2 + 1)$ such that:

$$C_{i,j} = \sum_{u=0}^{m_1-1} \sum_{v=0}^{m_2-1} A_{i+u,j+v} B_{u,v}, \quad i = 0, \ldots, n_1 - m_1 \text{ and } j = 0, \ldots, n_2 - m_2$$

Design an $O(n_1 n_2 (\log n_1 + \log n_2))$ algorithm for computing 2D discrete correlation. Would you use this algorithm in convolutional neural networks?

Solution 10. .

(a) Consider the polynomials $p(x) = p_0 + p_1 x + \cdots + p_{n-1} x^{n-1}$ and $q(x) = q_0 + q_1 x + \cdots + q_{n-1} x^{n-1}$, where $p_i = a_i$ for all i; $q_i = b_{m-1-i}$ when $i \leq m-1$; and $q_i = 0$ when $i \geq m$. Let $r(x) = r_0 + r_1 x + \cdots + r_{2n-2} x^{2n-2}$ be the product of $p(x)$ and $q(x)$. We have that

$$r_k = \sum_{i=0}^{k} p_i q_{k-i}, \quad k = 1, \ldots, 2n-2$$

where $p_j = q_j = 0$ when $j \geq n$. Let us look at the value of r_{m-1+k} when $k = 0, \ldots, n-m$. We have that

$$\begin{aligned} r_{m-1+k} &= \sum_{i=0}^{m-1+k} p_i q_{m-1+k-i} \\ &= \sum_{i=0}^{m-1+k} a_i q_{m-1+k-i} \end{aligned}$$

We have that $q_{m-1+k-i}$ is not set to zero only when $m-1+k-i \leq m-1$, which is equivalent to $k \leq i$ Therefore, continuing we get that

$$\begin{aligned} r_{m-1+k} &= \sum_{i=k}^{m-1+k} a_i q_{m-1+k-i} \\ &= \sum_{i=0}^{m-1} a_{i+k} q_{m-1-i} \\ &= \sum_{i=0}^{m-1} a_{i+k} b_i \\ &= c_k \end{aligned}$$

Summing up, to find c_k (the coefficients of the discrete correlation) we need to find the coefficients of $r(x)$, which can be done in $O(n \log n)$ time using the discrete FFT algorithm.

(b) Let $a_0, \ldots, a_{n_1-1} \in \mathbb{R}^{n_2}$ be the rows of A and $b_0, \ldots, b_{m_1-1} \in \mathbb{R}^{m_2}$ be the rows of B. Consider the following two arrays $\tilde{a} = [a_0 \,|a_1| \cdots |\, a_{n_1-1}]$ and $\tilde{b} = [b_0 \,|0_{n_2-m_2}|\, b_1 \,|0_{n_2-m_2}| \cdots |b_{m_1-1}|\, 0_{n_2-m_2}]$, where — denotes the concatenation operation and $0_{n_2-m_2}$ is a row vectors of $n_2 - m_2$ zeros. Let $\tilde{c}$ be the 1D discrete correlation of $\tilde{a}$ and $\tilde{b}$. By our construction, $\tilde{c}_k$ will equal to $C_{i,j}$, when $k = i \cdot n_2 + j$. Because we need to compute 1D discrete correlation, the complexity of our algorithm will be $O(n \log n)$, where $n = n_1 n_2$ (according to the part (a)). Using this algorithm in convolutional neural networks can be useful when the kernel size (filter size) $m_1 \times m_2$ is large enough, making the brute force solution too slow.

Problem 11. Imagine there is an algorithm called RandomSort that works as follows. Given an array of n distinct positive integers $a_1, \ldots, a_n, n \geq 3$. First, we initialize a counting variable $c = 0$. Each time, a pair of indices $i, j \in \{1, \ldots, n\}, i < j$, are drawn independently uniformly at random from the $\binom{n}{2}$ pairs. There are two cases that may happen:

- If $a_i < a_j$, then the program increments c by 1 and check if $c \geq T$ for some threshold $T > 0$: if yes, then the program terminates; otherwise, it moves forward to the next iteration;
- If $a_i > a_j$, then it swaps a_i and a_j, reset c to 0 , and go ahead to the next iteration.

(a) Find some upper bound for T (in terms of n) such that when RandomSort terminates, the probability that the array is sorted is at least $1 - \frac{1}{n}$.

(b) Argue that if we set $T = +\infty$, and suppose we could run the algorithm for an infinite number of iterations, the array must be sorted.

Solution 11. (a) We call a pair of indicies (i, j) inversion if $a_i > a_j$. Let X be the number of inversions at a time. Now, let (i, j) be a uniformly random pair, then we call an event bad if $a_i < a_j$ and good if $a_i > a_j$. Observe that the output is not sorted if T bad events occurs consecutively when $X > 0$. Thus,

$$\begin{aligned}
\Pr[\text{Output is unsorted}] &\leq \Pr[T \text{ bad event happens} \mid X > T] \\
&\leq \left(1 - \frac{X}{\binom{n}{2}}\right)^T \\
&\leq \left(1 - \frac{1}{n^2}\right)^T \\
&\leq e^{-T/n^2}.
\end{aligned}$$

Setting $T = n^2 \log n$ implies that fail probability is less than $1/n$.

(b) Notice that $e^{-T/n^2} \to 0$ as $T \to \infty$.

Problem 12. A Max-Cut of an undirected graph $G = (V, E)$ is defined as a cut $C_{\max} \subset V$ such that the number of edges between $C_{\max}$ and $V - C_{\max}$ is the maximum possible among all cuts. Consider the following iterative algorithm.

- Start with an arbitrary cut C.
- While there exists a vertex v such that moving v from one side of C to the other increases the number of edges crossing C, move v and update C.

(a) Does the algorithm terminate?

(b) Prove that if the algorithm terminates, then it yields a 2 approximation (that is the number of edges crossing $C_{\max}$ is at most twice the number crossing C).

Solution 12. (a) Notice that in each iteration the cut size is increasing. Since the cut size can be at most $\binom{n}{2}$, it will evantually terminate. In fact, the number of moves made by the algorithm is $O(n^2)$.

(b) Let's assume that the algorithm is terminated. Then, moving any of the vertices does not improve the cut size. Consider an arbitrary vertex v, and WLOG assume that $v \in C^-$ in one side of the cut (C^+ is the other side). Then, we have

$$\#\{(v, u) \in E \ : \ u \in C-\} \leq \#\{(v, u) \in E \ : \ u \in C^+\}.$$

Therefore, for each of the vertices at least half of their neighbors are on the other side. Thus, cut size is at least $\binom{n}{2}/2 \geq OPT/2$.

2 Bonus Problems

Problem 13. Chapter 4, exercise 8 of [2].

Solution 13. Assume for a contradiction that there are two distinct minimum spanning trees T_1 and T_2. Let e_1 be the smallest weight edge in $T_1 \setminus T_2$ and e_2 be the smallest weight edge $T_2 \setminus T_1$. Assume WLOG, $w(e_1) < w(e_2)$. Then, consider the graph $T_2 \cup \{e_1\}$ which contains a cycle. One can break the cycle by removing an edge with larger weight than e_1 (How?). This gives us another tree T_2' which has strictly less weight than T_2 and contradicts with the fact that T_2 is an MST.

Problem 14. Chapter 4, exercise 29 of [2].

Solution 14. Let $d_1, d_2, \ldots d_n$ be the sequence of remaining degrees. In each iteration we connect two vertices with maximum degrees, and decrease their remaining degrees by one. We iterate until degree sequence diminishes. We claim that if the given degree sequence is realizable, this greedy algorithm produces a graph instance. How? Think about the exchange argument.

Problem 15. Solve the problem of finding a clustering of maximum spacing. The problem and solution are described in Chapter 4.7 of [2]. Please spend some time thinking on the problem before reading the solution.

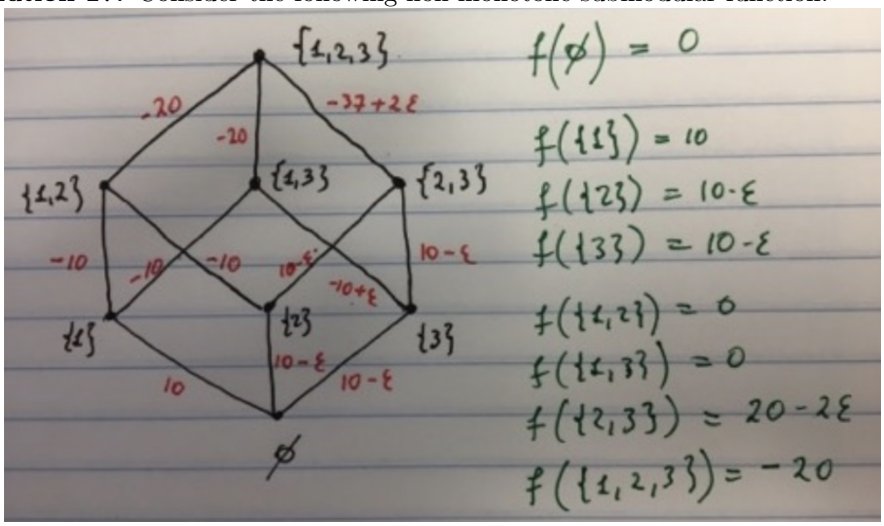
Solution 15. The solution is described in Chapter 4.7.

Problem 16. Prove Lemma 4.4 of [1]. As a bonus, consider proving Theorem 4.5 (under independent cascade model the influence function is submodular) and Theorem 4.6 (under linear Threshold model the influence function is submodular). Please spend some time thinking on the problem before reading the proofs.

Solution 16. See the paper.

Problem 17 (From Fall 2019)**.** Give an example of a submodular function such that the greedy algorithm discussed in the class fails to find a $(1-1/e)$ approximation for $\max_{|S|\leq k} f(S)$. Hint: f cannot be monotone.

Solution 17. Consider the following non-monotone submodular function.



Problem 18. Given an undirected graph $G = (V, E)$ with non-negative edge capacities $c : E \to R_+$, the cut capacity function $f : 2^V \to R_+$is defined by $f(S) = \sum_{e \in \delta(S)} c(e)$, where

$\delta(S) = \{e : e = (u, v) \in E$, and $(u \in S, v \in V \backslash S$ or $v \in S, u \in V \backslash S)\}$. Prove that the cut capacity function is submodular. Can we use this result to design a $(1 - 1/e)$ approximation algorithm for finding $\max_{|S| \leq k} f(S)$?

Solution 18. For any $A \subseteq V$ and any $u \in V$, let us define $\delta(u, A)$ be the set of edges with one endpoint at u and another at a vertex of A. To prove that f is submodular, consider any $S, T \subseteq V, S \subseteq T$ and any $x \in V \backslash T$. Then $f(S + \{x\}) - f(S) = -\sum_{e \in \delta(x,S)} c(e) + \sum_{e \in \delta(x, V \backslash S)} c(e)$, because adding x into S while change the $\delta(S)$ be removing edges of $\delta(x, S)$ and adding edges of $\delta(x, V \backslash S)$. Similarly, $f(T + \{x\}) - f(T) = -\sum_{e \in \delta(x,T)} c(e) + \sum_{e \in \delta(x, V \backslash T)} c(e)$. As $S \subseteq T$, then $\delta(x, S) \subseteq \delta(x, T)$, which implies that $-\sum_{e \in \delta(x,S)} c(e) \geq -\sum_{e \in \delta(x,T)} c(e)$. Additionally, if $S \subseteq T$ then $\delta(x, V \backslash T) \subseteq \delta(x, V \backslash S)$, which implies that $\sum_{e \in \delta(x, V \backslash S)} c(e) \geq \sum_{e \in \delta(x, V \backslash T)} c(e)$. Combining the last two results, we get that $f(S + \{x\}) - f(S) \geq f(T + \{x\}) - f(T)$, which proves the submodularity of f.

While f is indeed submodular, it is not monotone. Therefore, we cannot use the greedy algorithm directly.

Problem 19. Let $\Omega = \{v_1, v_2, \ldots, v_n\}$ be a set of vectors in $\mathbb{R}^d$. Consider the following set function $f : 2^\Omega \to \mathbb{R}, f(S) = \text{rank}(S)$, where $\text{rank}(S)$ is defined as the maximum number of linearly independent vectors of S. Prove that f is a submodular function.

Solution 19. Consider any $S, T \subseteq \Omega, S \subseteq T$ and any $v \in \Omega \backslash T$. We know that rank $\text{rank}(S) \leq \text{rank}(S \cup \{x\}) \leq \text{rank}(S) + 1$ (this is true for any x and S). There are two cases:

1. If $\text{rank}(S) = \text{rank}(S \cup \{x\})$, then x is linearly dependent with the vectors of S. Therefore, it will be linearly dependent with the vectors of T, implying that $\text{rank}(T) = \text{rank}(T \cup \{x\})$. We get that $0 = f(S \cup \{x\}) - f(S) \geq f(T \cup \{x\}) - f(T) = 0$
2. If $1 + \text{rank}(S) = \text{rank}(S \cup \{x\})$, then $f(S \cup \{x\}) - f(S) = 1$. We know that $\text{rank}(T \cup \{x\}) \leq \text{rank}(T) + 1$. Therefore $1 = f(S \cup \{x\}) - f(S) \geq f(T \cup \{x\}) - f(T)$

References

[1] David Kempe, Jon Kleinberg, and Eva Tardos. Maximizing the spread of influence through a social network. *Theory of Computing*, 11(4):105–147, 2015. URL: `http://www.theoryofcomputing.org/articles/v011a004`.

[2] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.