CSCI-SHU 220: Algorithms — Posted: March 3, 2024
Homework 2 — Due: 11:30pm (Shanghai time), March 14, 2024

This assignment has in total 70 base points and 10 extra points, and the cap is 70. Bonus questions are indicated using the $\star$ mark.

Please specify the following information before submission:

- Your Name: Yufeng Xu
- Your NetID: yx3038

Problem 1: Packing heavy unit intervals [15 pts]

Suppose we are given a set $P = \{x_1, \ldots, x_n\}$ of n points on the x-axis and a positive integer k. We say an interval on the x-axis is *k-heavy* if it contains at least k points in P. We consider *left-open right-closed unit intervals*, i.e., intervals of the form $I = (a, b]$ where $b - a = 1$ (for convenience, below we simply call them *unit intervals*). Our goal is to find a maximum number of k-heavy unit intervals on the x-axis that are *disjoint* from each other. For example, if $P = \{-\frac{2}{3}, 0, \frac{2}{3}\}$ and $k = 1$, then one optimal solution is $I_1 = (-1.6, -0.6]$, $I_2 = (-0.6, 0.4]$, $I_3 = (0.5, 1.5]$, which consists of three disjoint k-heavy unit intervals (note that the optimal solution is not necessarily unique). Design a greedy algorithm HeavyInt(n, P, k) to solve this problem, which should output the set of intervals you find. First describe your greedy strategy and prove its correctness. Then implement your algorithm by giving the pseudocode. Your algorithm is supposed to run in $O(n)$ time, assuming the points in P have already been sorted in the left-right order.

Solution. **Please write down your solution to Problem 1 here.**

Intuition of the algorithm: First, for $P = \{x_1, \ldots, x_n\}$, choose **the smallest** $t_1 (t_1 \geq k)$ such that $x_{t_1+k-1} - x_{t_1} < 1$. Take $(x_{t_1+k-1} - 1, x_{t_1+k-1}]$ as $I_1 = (a_1, b_1]$.

Repeat the process for $P_2 = \{x_l, \ldots, x_n\}$ such that x_l is the smallest element in P that is larger than b_1. Take $t_2 \geq l$ such that $x_{t_2+k-1} - x_{t_2} < 1$, then let $a_2 = \max(b_1, x_{t_2+k-1} - 1)$, $I_2 = (a_2, a_2 + 1]$. **Note we are choosing the smallest non-conflicting interval with this strategy.**

Repeat until for some P_{c+1} that we can no longer find t_{c+1} such that $x_{t_c+k-1} - x_{t_c} < 1$. Then the total number of intervals is c.

Proof of correctness: . Assume the intervals chosen by this greedy algorithm are $I_{g_1}, I_{g_2}, \ldots, I_{g_c}$, we want to show for any $j \in \{0, 1, \ldots, c\}$, $\exists$ an optimal solution that contains $I_{g_1}, \ldots, I_{g_j}$.

When $j = 0$, an optimal solution doesn't have to contain any intervals among $I_{g_1}, ..., I_{g_c}$, so this statement is true. Now, assume for some j, $\exists$ an optimal solution that contains $I_{g_1}, \ldots, I_{g_j}$. Assume the $j + 1^{th}$ chosen interval in this optimal solution is I_x, if we replace I_x with $I_{g_{j+1}}$, by definition, $I_{g_{j+1}}$ is the interval with the smallest right end after choosing $I_{g_1}, \ldots, I_{g_j}$, hence the right end of $I_{g_{j+1}}$ is smaller than that of I_x. In other words, replacing I_x with $I_{g_{j+1}}$ doesn't interfere the selection of the subsequent intervals. The modified solution is still optimal.

By induction, when $j = c$, $\exists$ an optimal solution that contains $I_{g_1}, \ldots, I_{g_c}$. By definition, after making this selection, we can no longer choose another interval, hence this optimal solutions is $\{I_{g_1}, \ldots, I_{g_c}\}$, and the greedy algorithm is optimal.
Python implementation:

```
def HeavyInt(n, P, k):
    t = 0
    intervals = []
    while t + k - 1 < n:
        while t + k - 1 < n and P[t + k - 1] - P[t] >= 1:
            t += 1
        if t + k - 1 < n:
            if len(intervals) > 0:
                left = max(intervals[-1][1], P[t + k - 1] - 1)
            else:
                left = P[t + k - 1] - 1
            intervals.append((left, left + 1))
            while t + k - 1 < n and P[t] <= intervals[-1][1]:
                t += 1
    return intervals
```

Problem 2: Sorting with arbitrary swaps $[10 + 10 + 10^\star$ pts$]$

Given an array $A[1 \ldots n]$ of distinct numbers, we want to sort it in increasing order. However, we are only allowed to change A by swapping two elements (not necessarily) in A. Specifically, we are provided an oracle Swap. If we call Swap(i, j), then $A[i]$ and $A[j]$ will be swapped in A.

(a) Design a simple algorithm that sorts A by calling the Swap oracle at most $n - 1$ times. Describe the basic idea and give the pseudocode, and analyze the number of oracle calls. Your algorithm is supposed to run in $O(n \log n)$ time (and call Swap at most $n - 1$ times), assuming each oracle call takes $O(1)$ time.

(b) Next, we consider the problem of sorting A using *minimum* number of calls of the Swap oracle. Prof. Greed gives the following greedy algorithm for this problem. Let $\#\mathsf{Inv}(A)$ denote the number of inversions in A. The algorithm simply repeats the following step until A is sorted: find indices $i, j \in \{1, \ldots, n\}$ such that swapping $A[i]$ and $A[j]$ makes $\#\mathsf{Inv}(A)$ decrease the most, and then call Swap(i, j). Give a counterexample for which this algorithm fails to give an optimal solution (and briefly argue why it is a counterexample). To get the full credit, your example should make the algorithm fail no matter how it breaks the ties.

(c*) Design an algorithm that sorts A using minimum number of Swap calls. Describe the basic idea and give the pseudocode. Prove the correctness of your algorithm (i.e., it successfully sorts A and the number of Swap calls used is minimum). Your algorithm is supposed to run in $O(n^2)$ time or faster.

Solution. **Please write down your solution to Problem 2 here.**
(a) **Basic idea:** We first use a quick-sort-like algorithm to obtain the rank of each element in A,

which takes $O(n \log n)$ time; then we make at most $n-1$ swaps to directly put each element in its correct position, which takes $O(n)$ time.

Pseudocode:

```
def simpleSort(n, A):
    rank = {}
    def SWAP(i, j):
        A[i], A[j] = A[j], A[i]
    def indexer(A, add, rank):
        if len(A) == 0:
            return
        pivot = random.choice(A)
        L = []
        R = []
        for each in A:
            if each < pivot:
                L.append(each)
            elif each > pivot:
                R.append(each)
        rank[pivot] = add + len(L)
        indexer(L, add, rank)
        indexer(R, add + len(L) + 1, rank)
    indexer(A, 0, rank)
    def swapSort(n, A):
        swap_cnt = 0
        for i in range(n - 1):
            while i != rank[A[i]]:
                swap(i, rank[A[i]])
                swap_cnt += 1
        print(swap_cnt)
    swapSort(n, A)
```

Analysis of number of oracle calls: After obtaining the rank of all of the elements, for each swap, we put at least 1 element in place and will never put one element was already in place out of place. Therefore, in at most $n-1$ swaps, we will put $n-1$ elements in place, so the remaining one element will also be in place. In other words, all n elements will be in place after $n-1$ calls, so the number of times we call oracle swap is at most $n-1$.

(b)**A counter example:** $A = [6, 5, 1, 3, 2, 4]$.

Illustration: Assume $\exists i < j$ such that $A[i] > A[j]$, if we swap $A[i]$ and $A[j]$, the decrease of #Inv is equal to $1 + |\{k | i < k < j, A[j] < A[k] < A[i]\}|$. Therefore, in order to make #Inv decrease the most, we should swap 6 and 2 in the first place (#Inv decreases by 3). However, sorting the swapped array $A' = [2, 5, 1, 3, 6, 4]$ takes at least 5 swaps, so in total it takes 6 swaps to sort this array with the greedy algorithm.

Alternatively, this array can be sorted by swapping 2 and 5 in the first place(#Inv decreases by 2), and sorting the swapped array $A' = [6, 2, 1, 3, 5, 4]$ takes only 3 steps ($[6, 2, 1, 3, 5, 4] \to [1, 2, 6, 3, 5, 4] \to [1, 2, 3, 6, 5, 4] \to [1, 2, 3, 4, 5, 6]$), so in total it only takes 4 swaps to sort this array. Therefore, in this case, the greedy algorithm is not optimal.

(c) We apply exactly the same algorithm in 2(a) and show it's optimal in terms of # swaps.
Analysis of minimum number of swaps: We define a "cycle" as follows: $C_i = \{A_{i_1}, \ldots, A_{i_{k_i}}\}$,

$idx(a)$ as the actual index of a in A, $rank(a)$ as the rank of a in A, then $rank(A_{i_1}) = idx(A_{i_2}), \ldots,$ $rank(A_{i_{k_i}-1}) = idx(A_{i_{k_i}}), rank(A_{i_{k_i}}) = idx(A_{i_1})$. Any unsorted array A can be decomposed into several cycles. **Any swap between the elements in two cycles is meaningless because it will not put any element in place or increase the number of cycles.** Therefore, the number of swaps needed to sort A is equal to the sum of the number of swaps needed to sort each cycle.

Next, we prove by induction that in order to sort a cycle with $|C_i| = k_i$, we need to perform at least $k_i - 1$ swaps. For $k_i = 2$, we have $C_i = \{A_{i_1}, A_{i_2}\}$, where $rank(A_{i_1}) = idx(A_{i_2}), rank(A_{i_2}) = idx(A_{i_1})$, we only need to swap A_{i_1} and A_{i_2}. Hence we need to make $1 = 2 - 1$ swap to sort this cycle.

Assume for $|C_i| = k$, we need to make $k - 1$ swaps to make the cycle sorted. For $|C_i| = k + 1$, any swap within the array will only put at most 1 element in place. Consider making a swap that puts one element in place (swap A_{i_j} and $A_{i_{j+1}}$), now we have $idx(A_{i_j}) = rank(A_{i_j})$, and $rank(A_{i_{j-1}}) = idx(A_{i_{j+1}}), rank(A_{i_{j+1}}) = idx(A_{i_{j+2}})$. In other words, we obtained an element in place and a new cycle C_i' with $|C_i'| = k$, which takes at least $k - 1$ swaps to sort. Hence $|C_i|$ takes at least k steps to sort.

Finally, we prove that our algorithm obtains this optimal solution. For each i, our algorithm keeps swapping $A[i]$ and $A[rank(A_i)]$ if $i \neq rank(A[i])$. This is equivalent to swap A_{i_j} and $A_{i_{j+1}}$ where $A_{i_j}, A_{i_{j+1}} \in C$, $rank(A_{i_j}) = idx(A_{i_{j+1}})$ Consequently, we have $A_{i_{j+1}}$ at place i and $rank(A_{i_{j+1}}) = idx(A_{i_{j+2}})$ and $rank(A_{i_{j-1}}) = idx(A_{i_{j+1}})$. Therefore, when at last $i = rank(A[i])$, we finish sorting a cycle C with $|C| - 1$ swaps. For any sorted cycle C, we have $i = rank(A_i)$ for all $A_i \in C$ and will not swap at all. Therefore, this algorithm sorts A with only $\Sigma_C(|C| - 1) = n - \#$cycles swaps, which is optimal.

Pseudocode: See 2(a).

Problem 3: Longest doubling subsequence [15 pts]

Recall that an array (or sequence) A of real numbers is *doubling* if $A[i] \geq 2A[i-1]$ for all $i \geq 2$ (thus any array of length 1 is doubling). Given an array A, we want to find a longest subsequence of A that is doubling. For example, if $A = [7, 1, 3, 8, 2, 4, 5, 6, 9]$, then a longest doubling subsequence of A is be $[1, 2, 4, 9]$, which is of length 4. Design an algorithm LDS(n, A) that returns a longest doubling subsequence A' of the array A (where n is the size of A). Describe the basic idea of your algorithm and give the pseudocode. Briefly justify its correctness and analyze its time complexity. Your algorithm should run in $O(n^2)$ time or faster.

Solution. **Please write down your solution to Problem 3 here.**

Intuition of algorithm: We define the maximum length of a subsequence ending at A_i as L_i. $P_i = \{i_1, \ldots, i_k\}$ such that $\forall j \in P_i, j < i, 2A_j \leq A_i$. Let $L_i = \max_{j \in P_i}(L_j) + 1$, $prev_i = \text{argmin}_{j \in P_i}(L_j)$. After we obtain all L_i for $\forall i \in \{1, 2, \ldots, n\}$, we say that the maximal length of any LDS is $max_{i \in \{1, \ldots, n\}} L_i$, and the corresponding LDS is obtained by checking $prev_i$ recursively.

Pseudocode:

```
def LDS(n, A):
    length = []
    last = []
    length.append(1)
    last.append(None)
    for i in range(1, n):
        max_len = 0
        tmp = None
        for j in range(0, i):
            if length[j] > max_len and A[i] >= 2 * A[j]:
                max_len = length[j]
                tmp = j
        length.append(max_len + 1)
        last.append(tmp)
    tail = length.index(max(length))
    res = []
    while tail != None:
        res.append(A[tail])
        tail = last[tail]
    return res[::-1]
```

Proof of correctness: Assume an optimal solution ending at A_i contains $\{A_{i_1}, \ldots, A_j, A_i\}$, then $\{A_{i_1}, \ldots, A_j\}$ is an optimal solution that ends at A_j. Therefore, in order to obtain the optimal solution that ends at A_i, we only need to retrieve the best solution among the optimal solutions at ends at $A_{i_1}, \ldots, A_{i_k}$, where $\forall j \in \{i_1, \ldots, i_k\}, j < i, 2A_j \leq A_i$.

Time complexity analysis: For any i, we need to check $j = 1, \ldots, i-1$ to see if $2A_j \leq A_i$ and the optimal solution that ends at A_j. For each j, the checking time is $O(1)$; for $j = 1, \ldots, i-1$, the total time is $O(n)$; for $i = 1, 2, \ldots, n$, the total time is $O(n^2)$. Hence the time complexity of this algorithm is $O(n^2)$.

Problem 4: Removing the numbers optimally [20 pts]

Given a sequence of (positive) numbers, we want to remove the numbers from the sequence one by one. When removing one number x, we gain a score equal to l^2xr^2 where l is the number to the left of x in the current sequence ($l = 1$ if x is the leftmost number in the current sequence) and r is the number to the right of x in the current sequence ($r = 1$ if x is the rightmost number in the current sequence). For example, suppose the given sequence is $(2, 9, 12, 3)$. If we remove 12 first, the score we gain is $9^2 \times 12 \times 3^2 = 8748$, and the sequence becomes $(2, 9, 3)$. Now if we remove 9 from the sequence, then the score is $2^2 \times 9 \times 3^2 = 324$, and the sequence becomes $(2, 3)$. Next if we remove 3, then the score is $2^2 \times 3 \times 1^2 = 12$, and the sequence becomes (2). Finally we remove 2, and the score is $1^2 \times 2 \times 1^2 = 2$. The total score we gain is $8748 + 324 + 12 + 2 = 9086$. Our goal is to find an order to remove the numbers from the given sequence such that the total score we gain is maximized. Formally, design an algorithm REMOVE(n, A) where A is an array of n positive numbers; the algorithm returns the maximum total score we can gain if the given sequence is A (for convenience, you are not required to return the optimal order for removing the numbers). Describe the basic idea of your algorithm and give the pseudocode. Briefly justify its correctness and analyze its time complexity. Your algorithm should run in $O(n^3)$ time or faster.

Solution. **Please write down your solution to Problem 4 here.**
Idea of the algorithm: In the first place, we augment the array A to a new array B, $B = (1, A_1, \ldots, A_n, 1) = (B_1, B_2, \ldots, B_{n+1}, B_{n+2})$. We need to remove $B_2, \ldots, B_{n+1}$, and the score obtained by removing B_j is equal to $B_{j-1}^2 \cdot B_j \cdot B_{j+1}^2$ for any $j \in \{2, \ldots, n+1\}$.

Now, let $1 \leq l < r \leq n+2$. Consider the optimal score obtained after removing all elements in $\{B_{l+1}, \ldots, B_{r-1}\}$ $f(l,r)$. Suppose the last element we remove from $\{B_{l+1}, \ldots, B_{r-1}\}$ is B_k where $l < k < r$, then the optimal score can be represented as:

$$f(l,r) = \max_k\{f(l,k) + f(k,r) + B_l^2 \cdot B_k \cdot B_r^2\}$$

The boundary case is that if $r - l = 2$, $f(l,r) = B_l^2 \cdot B_{l+1} \cdot B_r^2$.

Pseudocode:

```
def Remove(n, A):
    memory = {}
    def helper(B, l, r):
        if memory.get((l, r), 0) > 0:
            return memory[(l, r)]
        if r - l == 2:
            return B[l] ** 2 * B[l + 1] * B[r] ** 2
        maxn = 0
        for i in range(l + 1, r):
            tmp = helper(B, l, i) + helper(B, i, r) + B[l] ** 2 * B[i] * B[r] ** 2
            if tmp > maxn:
                maxn = tmp
        memory[(l, r)] = maxn
        return maxn
    B = [1] + A + [1]
    helper(B, 0, n + 1)
    return memory[(0, n + 1)]
```

Justification of correctness: We prove by induction that this algorithm gives the optimal result. When $n = 1$, $f(1,3) = B_2 = A_1$, this algorithm gives the best result.

Assume when $n = 1, 2, \ldots, k$, this algorithm gives the optimal result. When $n = k+1$, we know in $A = \{A_1, \ldots, A_{k+1}\}$, we must choose one element A_j to remove at the last, and the score obtained is equivalent to the score obtained by removing $A_1, \ldots, A_{j-1}$(denote as L) + the score by removing $A_{j+1}, \ldots, A_n$(denote as R)+A_j, where the first part and the second part are independent of each other, and A_j is not dependent on L and R(but L and R are dependent on A_j). Therefore, $\max_{(L,R,A_j)}\{L + R + A_j\} = \max_{A_j}\{\max_{A_j}\{L\} + \max_{A_j}\{R\} + A_j\}$, where by our assumption, $\max_{A_j}\{L\}$ and $\max_{A_j}\{R\}$ can be obtained by our algorithm. Therefore when $n = k+1$, the global optimal can also be obtained by this algorithm. By induction, this algorithm works for $\forall n \in \mathbb{N}^*$.

Analysis of time complexity: In this algorithm, we need to traverse all the possible l, r, k without redundancy, where $l < r < k$. The number of all possible combinations (l, k, r) is equal to $\binom{n+2}{3} = \frac{(n+2)(n+1)n}{6} = O(n^3)$. The time complexity for each combination is $O(1)$, hence the time complexity is $O(n^3)$.