CSCI-SHU 220: Algorithms @ NYU Shanghai — Last Updated: November 9, 2020
Homework 4 — Due Date: November 23, 2020, 11:00pm

You are allowed to discuss with others but not allow to use references other than the course notes and reference books. Please list your collaborators for each questions. Write your own solutions and make sure you understand them.

There are 55 marks in total (including the bonus). The full mark of this homework is 50.

Enjoy :).

Please specify the following information before submission:

- Your Name: Zhenming Wang
- Your NetID: zw1806
- Collaborators:

Problem 1: Subset sum [10 marks]

Given a set A of integers, devise an algorithm based on dynamic programming to determine whether A can be partitioned into two subsets so that the sums of the elements in both subsets are equal.

Example 1: with $A = \{1, 2, 3, 6\}$ the answer is yes,

Example 2: with $A = \{1, 2, 6\}$ the answer is no.

Solution. **Please write down your solution to Problem 1 here.**
To determine whether the original set A could be partitioned into two subsets with equal sum, we could achieve this by finding a subset whose sum is equal to half of the whole set.
We would use a 2-D DP table *subset* of size length(A) $\times$ sum(A), and in each cell *subset*[i][j] we store the result of whether there exists a subset in A[:i] whose sum equals to j, *True* of *False*.

$$\text{subset[i,j]} = \text{subset[i-1,j]} \text{ or } \text{subset[i-1,j-}A_i\text{]}$$

```
def equalSub(A,S,n):
    if S%2 != 0:
        return False
    else:
        //divide the sum of the whole array
        sub_sum = S // 2

        //intialize the DP table
        subset = ([[False for i in range (sub_sum)+1] for j in range(n+1)])

        //when the sum is 0, then the answer is true
        for i in range(n+1):
            subset[i][0] = True
```

```
        //when the subset contains no element,
        //all the subset sum greater than 0 should have an answer False
        for i in range(1, sub_sum+1):
            subset[0][i] = False
        //main loop
        for i in range(1,n+1):
            for j in range(1,sub_sum+1):
                if A[i-1] <= j:
                    subset[i][j] = (subset[i-1][j] or subset[i-1][j-A[i-1]])
                elif A[i-1]>j:
                    subset[i][j] = subset[i-1][j]

        return subset[n][sub_sum]
```

Problem 2: Knapsack [10 marks]

Assume that there are n objects, and the i-th object weights $w_i > 0$ kilograms and has value v_i. Here n and v_i are both positive integers with $n > v_i$. With a knapsack of capacity $W > 0$ kilograms, design an algorithm to find the maximum total value of the objects with which we can fill the knapsack. You will get full marks if the algorithm is of complexity $O(n^3)$.

Solution. **Please write down your solution to Problem 2 here.**
Since it's given that for each object who has value v_i, $n > v_i$, which means

$$n > V, V = max\{v_i\}$$

The loose upper-bound value we could get from the set of n objects is therefore n^2. So, we would use a 2-D DP table KS of size $n \times n^2$, with each cell KS[i][v] represents the minimum weight we could use from objects set $\{A_1 \cdots A_i\}$ to get to value v.

$$\text{KS[i][v]} = min\begin{cases} KS[i-1][v] \\ w_i + KS[i-1][v-v_i] \end{cases}$$

After we filled all the cell in KS, we perform a linear scan on KS[n], and find the greatest v that is less than W.

```
    def knapsack(w,v,n,W):

        \\intialize table of size  n×n^2
        KS = [[0 for j in range(n^2+1)] for i in range(n+1)]

        //with 0 objects, we could not reach any of the values greater than 0
        for i in range(1,n^2+1):
            KS[0][i] = float('inf')

        for i in range(1,n+1):
            for j in range(n^2+1):
                if v[i-1] <= j:
                    KS[i][j] = min(KS[i-1][j], w[i-1]+KS[i-1][j-v[i-1]])
```

```
            else:
                KS[i][j] - KS[i-1][j]

    \\solution lies in the nth row, so we traverse to find biggest v

    for i in range(n^2+1):
        if KS[n][i] > W:
            return i-1
        else:
            continue
```

Problem 3: Sorting [10 marks]

Given a sequence of numbers, $A = (a_1, \ldots, a_n)$, we will sort A in increasing order. Assume that we can insert any element of A into any position. Design an algorithm based on dynamic programming to find the minimum number of insertions that we need to sort A.

Example: with $A = (2, 3, 5, 1, 4, 7, 6)$ we need 3 insertions. First insert 1 before 2. Then insert 4 between 3 and 5. Finally insert 7 after 6 (or alternatively insert 6 before 7).

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

We first observe that moving an element would not change the relative order of elements in the sequence other than the one being moved. In order to minimize the number of movements (insertions), we consider the *longest non-decreasing subsequence*, and by only moving the element that is not in this sequence, it's trivial that we would get a sorted sequence, with all the elements that were already in the sequence have been sorted relative to each other.
Therefore, to find the minimum number of insertions to sort A, it suffices to find the length of the longest non-decreasing sub-sequence, and compute the difference between the length of the sequence and the length of the longest non-decreasing sub-sequence. We would use a 1-D DP table L to store the longest non-decreasing sub-sequence starting from A[i].

$$\text{L[i]} = 1 + max\{L[j], j < i \leq n, A[i] \geq A[j]\}$$

```
def longestnds(A,n):
    if n <= 1:
        return n
    else:
        \\initiate DP table
        DP = [1 for i in range(n)]

        for i in range(1,n):
            for j in range(i):
                if A[i] ≥ A[j] and DP[i]<DP[j] + 1:
                    A[i] = A[j] + 1

        return n–max(DP)
```

Problem 4: Covering Set [10 marks]

Given an undirected graph $G = (V, E)$, a subset $S \subseteq V$ is a covering set if for every vertex $v \in V \setminus S$, there exists $u \in S$ such that $uv \in E$ (i.e. every vertex $v \in V \setminus S$ has a neighbor in S). We are interested in finding a covering set of minimum total weight, but this problem is NP-hard in general graphs. Show that this problem is easy in trees.

- Input: A tree $T = (V, E)$ in the adjacency list representation, a weight w_v for each vertex $v \in V$.
- Output: A covering set $S \subseteq V$ that minimizes the total weight $\sum_{v \in S} w_v$.

Design an efficient algorithm to solve this problem. Prove the correctness and analyze the time complexity.

Solution. **Please write down your solution to Problem 4 here.**
The collection of Subproblem: for each vertex v in the tree, consider $W(v)$, that is

the minimal weighted covering set in the sub tree rooted at v

and to determine $W(v)$, there are two cases:

include v in the covering set
not including v in the covering set

We build a DP table of size $n \times 2$, with 2 columns named *true* and *false*, representing the weight of the minimal weight of the covering set in either of the two cases above.
Base case: if v is a leaf; DP[v][true] = w_v, DP[v][false] = 0

```
DP = [0 for i in range(2)] for j in range(|V|)

//first find the leaf node, and set their corresponding value in DP table
for i in range(V):
    if V[i] is not connected with any other node:
        DP[i][0] = V[i].weight
        DP[i][1] = 0

//the parameters below should all be their index in the actual list of nodes
coveringSet(v):
    for c in v's children:
        coveringSet(c)


    //if v included, then we choose the minimal one among whether to include
    // v's children or not
    DP[v][0] = w_v + sum for c in children(v) of min (DP[c][0], DP[c][1])


    //if v is not included, we must include all of v's children
    DP[v][1] = sum(DP[c][0] for c in v's children)
```

```
if DP[v][0] < DP[v][1]:
    add v to the covering vertex set
else:
    for c in v's children:
        add c to the covering vertex set
```

Proof. Our algorithm works in a way from the leaf node back to the root node. For each node, there are two choices, include it or not. For the leaf nodes, we have filled in the two base cases in advance using a traversal. Now, with our base case ready, we could go back up from leaf nodes back all the way to the root.

Consider the optimal solution W(v), the minimum total weight of a covering subset rooted at v. If we do not use v, then we would have to use $\sum_{\text{c in chlidren of v}} W(c)$, where $W(c) = \text{DP[c][0]}$, since we must use every c among the children of v, or we would have two adjacent level left not covered. If we use v, then we would look at each children c of v, and pick the minimal one among DP[c][0] and DP[c][1], since again we could choose to include c or not.And each value in the DP table is itself an optimal value, so we are still making the optimal decision based on the decision that we include v . After that, we would take the minimal one among DP[v][0] and DP[v][1], so we again obtain an optimal solution. □

Time Complexity: The algorithm would take $O(|V|)$ time.
Finding the leaf nodes would take $O(|V|)$ time, and solve the problem using DP table would take $O(|V|)$ time, since each node is visited only when we consider their parent, so each node is visited constant time.

Problem 5: Selection [10 marks + 5 marks]

Let $x = (x_1, \ldots, x_m)$ and $y = (y_1, \ldots, y_k)$ be two *sorted* sequences, i.e., they satisfy

$$x_1 \le x_2 \le \cdots \le x_m, \quad y_1 \le y_2 \le \cdots \le y_k. \tag{1}$$

There are $\binom{m}{k}$ subsequences of x of length k, and each subsequence has $k!$ rearrangements. Denote by X the set of those rearrangements of all length-k subsequences. Note that, for each element $x' = (x_{i_1}, \ldots, x_{i_k})$ of X, the sequence x' might not be sorted, that is, it is possible that $x_{i_j} > x_{i_{j+1}}$ for some j. We can compute the squared Euclidean distance between y and x',

$$d(y, x') := \sum_{j=1}^{k} (y_j - x_{i_j})^2. \tag{2}$$

Consider the following optimization problem

$$\min_{x' \in X} \; d(y, x'). \tag{3}$$

(a) (**Bonus**: 5 marks) Prove that there some element of X which is an optimal solution to (3) and is also a sorted sequence.

(b) (10 marks) Design an algorithm based on dynamic programming to find an optimal solution to (3) (not just the optimal value).

Solution. **Please write down your solution to Problem 5 here.**

(a) *Proof.* First, we would assume that there exists some optimal solution X', such that X' is not in an sorted fashion, which means, there exists at least one pair of local inversion $x_{i_j}$and$x_{i_{j+1}}$, such that $x_ij > x_ij + 1$. Now, let us exchange the position of x_{i_j} and $x_{i_{j+1}}$ for the sake of restoring the order.

$$\text{Before exchange, original local sum of squared distance}(y_{j+1} - x_{i_{j+1}})^2 + (y_j)^2 \quad \text{(equation 1)}$$

$$\text{After exchange, local sum of squared distance}(y_{j+1} - x_{i_j})^2 + (y_j - x_{i_{j+1}})^2 \quad \text{(equation2)}$$

We take the difference between the two sums, and evetually reach at the following equation:

$$\text{equation1 - equation2} = 2(x_{i_{j+1}} - x_{i_j})(y_j - y_{j+1}) > 0$$

Therefore, after swap, we obtain a better solution, so the sorted sequence is an optimal solution. □

(b) Since we are looking for an one-to-one relationship between y_i and x'_i, and according to (a), it suffices to find an optimal solution in sorted order. So for each element in X, there are two cases, take it in the solution subset or not.
We could write the recurrence relation for the sum of the squared distance as

$$d(y_i, x'_i) = min(d(y_i, x'_{i-1}), d(y_{i-1}, x'_{i-1}) + distance(y_i, x_i)))$$

To utilize dynamic programming, we would build a DP table of size $m \times k$, and we find every corresponding x'_i for each y_i. Say, for y_i, we pick x_j as its opponent, then we would fill DP[i][j] to be DP[i-1][last answer's index] $+(y_i - x_j)^2$.
How do we determine for each y_i which x_j to pick? Note that we are looking for a solution in sorted order, and $y_1 \cdots y_k$ is already in sorted order, so every x'_i we choose must locate in original X in sorted order. We would have an array called **solution** which records the x'_i 's index in the original array, and during the for loop on y_i, we look for the appropriate x'_j in the range of [**solution[-1]**, $m-(k-(i+1))$], the **solution[-1]** means that we are only looking for x'_j after previous choice of x'_{j-1}, while $m-(k-(i+1))$ is actually reserving enough set of x to choose from for the succeeding $y_{i+1}, \cdots, y_k$, or we would possibly have not enough x to choose from in order to preserve the property of the X', that it is in sorted order.
We pick the appropriate x'_j by computing their distance with current y_i, and take the x' with the minimum distance, and we append its index in the X sequence to **solution**.

```
def minimumSumOfDistance(X,Y,m,k):

    \\initiate DP table of size m×k
    DP = [[0 for i in range(m)] for j in range(k+1)]

    \\pre filled the first row of DP
```

```
\\ when Y does not contain anything
\\also convenient for later backtracking
for i in range(m):
    DP[0][i] = 0

\\initiate the lowerbound  for picking
\\the appropriate x'
lower_bound = 0

\\initiate the solution array containing the picked x's index
solution = []

\\main loop
for i in range(1,k+1):

    \\initiate the choice upperbound for picking x'
    \\should be (m-(k-(i+1)))

    upperbound = (m - (k- (i+1)))

    min_distance = 0  \\ min squared distance indicator
    min_index = none  \\ the x's index with current minimal distance
    for j in range(lower_bound, upperbound+1):
        distance<- compute distance between Y[i] and X[j]
        if distance < min_distance:
            min = distance
            min_index = j

    \\add min_index to solution array
    solution.add(min_index)

    \\we have found the appropriate x, so update DP[i][min_index]
    \\By finding previous sum of squared distance
    \\recall that each filled cell records current sum of squared distance
    \\and the solution for last question lies in
    \\DP[i-1][solution[-1]]

    DP[i][min_index] = DP[i-1][solution[-1]] + min_distance


\\the final sum of squared distance
minimal_sum = DP[k][solution[-1]]

\\the final set of x'
subset = []
for i in range(len(solution)):
    subset.append(X[solution[i]])
```