COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 10

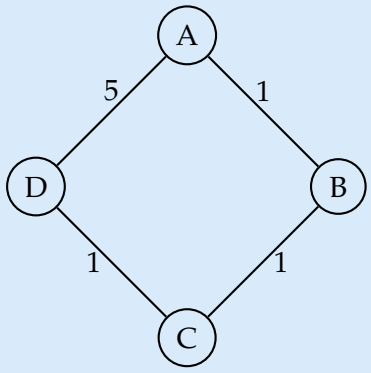
Announced on: Oct 10 (Thurs)

- This tutorial sheet contains problems in dynamic programming (DP). When presenting your solutions, please define the DP table clearly. Don't forget to write the base case. The correctness and running time arguments can be brief (1-2 sentences).
- Problems marked with ($\star$) will not be asked in the tutorial quiz.

Note: For some of the problems, we've only provided the main ideas and not the full solutions. We expect you to fill the gaps by yourself. If there are any doubts, you can contact your tutorial TA, head TA or the instructor.

1. Recall that once the subproblem solutions stabilize in the Bellman-Ford algorithm (with $L_{k+1,v} = L_{k,v}$ for every destination v), they remain the same forevermore (with $L_{i,v} = L_{k,v}$ for all $i \geqslant k$ and $v \in V$). Is this also true on a per-vertex basis? That is, is it true that, whenever $L_{k+1,v} = L_{k,v}$ for some $k \geqslant 0$ and destination v, then $L_{i,v} = L_{k,v}$ for all $i \geqslant k$? Provide either a proof or a counterexample.

   It is **False**. Consider the following counterexample, where we aim to find the shortest distance path starting from vertex A.

   The initial distance estimate for vertex D after the first iteration is $L_{1,D} = 5$, and it stays the same after the second iteration as well ($L_{2,D} = 5$). However, after the third iteration, the distance to vertex D updates to $L_{3,D} = 3$.

2. You are given a *strongly connected* directed graph $G = (V, E)$, which means that there is a directed path between every ordered pair of vertices. Each edge $e \in E$ of this graph is either colored red or blue. Design an algorithm to figure out whether there exists a cycle in G with

strictly more red edges than blue edges. Clearly state the running time of your algorithm.

The red edges can be given edge weight of -1 and the blue edges be given edge weight of 1. Thus, the problem reduces to finding a *strictly negative weight* cycle in G. We can use Bellman-Ford to check whether there exists a cycle of negative weight in the graph, and if it does, find one of these cycles.

Criterion for presence of a negative weight cycle reachable from source vertex v: After $(|V|-1)^{th}$ iteration, if we run the algorithm for one more iteration, and it performs at least one more relaxation, then G contains a negative weight cycle that is reachable from v; otherwise, such a negative weight cycle does not exist. Since G is *strongly connected*, all the vertices are reachable from a source vertex v.

Time complexity: Same as that of Bellman-Ford algorithm $O(|V|.|E|)$

3. Consider the following parenthesization puzzles:

   a) *Parenthesize* $6 + 0 \cdot 6$ *to maximize the outcome*.

      Incorrect answer: $6 + (0 \cdot 6) = 6 + 0 = 6$.

      Correct answer: $(6 + 0) \cdot 6 = 6 \cdot 6 = 36$.

   b) *Parenthesize* $0.1 \cdot 0.1 + 0.1$ *to maximize the outcome*.

      Incorrect answer: $0.1 \cdot (0.1 + 0.1) = 0.1 \cdot 0.2 = 0.02$.

      Correct answer: $(0.1 \cdot 0.1) + 0.1 = 0.01 + 0.1 = 0.11$.

   The input to your algorithm is a sequence $x_0, o_0, x_1, o_1, \ldots, x_{n-1}, o_{n-1}, x_n$ of $n+1$ real numbers $x_0, x_1, \ldots, x_n$ and n operators $o_0, o_1, \ldots, o_{n-1}$. Each operator o_i is either addition $(+)$ or multiplication $(\cdot)$. Design a polynomial-time dynamic programming algorithm for finding the optimal (maximum-outcome) parenthesization of the given expression, and analyze the running time.

ALGORITHM 1: Maximum outcome after parenthesizing the given expression

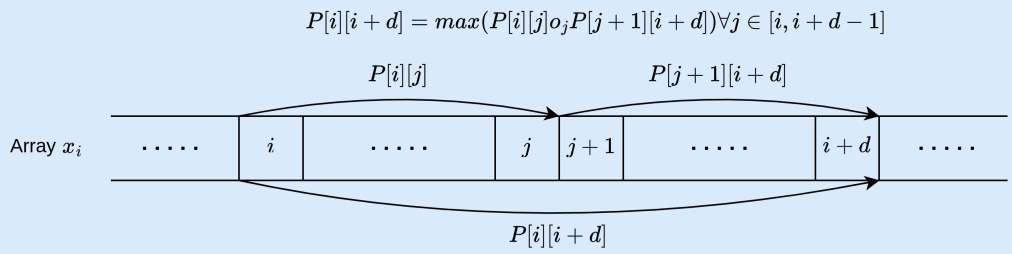
Input: A list $[x_0, x_1, \ldots, x_n]$ of $n+1$ real numbers and a list $[o_0, o_1, \ldots, o_n]$ of n operators.
Output: The maximum outcome after parenthesization of the given expression.

Let M and m be two $(n+1) \times (n+1)$ matrices such that the $M[i][j]$ (initialized to all $-\infty$) represents the maximum outcome when the expression is evaluated in the contiguous sequence $x_i, o_i, x_{i+1}, o_{i+1}, \ldots, x_j$ and $m[i][j]$ (initialized to all ∞) represents the minimum outcome when the same contiguous expression is evaluated.
Base condition: $M[i][i] = m[i][i] = x_i \; \forall i \in [0, n]$ (as expression is the real number x_i with no operator)
for $d \leftarrow 1$ **to** n **do**
    for $i \leftarrow 0$ **to** $n - d$ **do**
        for $j \leftarrow i$ **to** $i + d - 1$ **do**
            if o_j *is + operator* **then**
                $M[i][i+d] \leftarrow max(M[i][i+d], M[i][j] + M[j+1][i+d])$
                $m[i][i+d] \leftarrow min(m[i][i+d], m[i][j] + m[j+1][i+d])$
            else
                $M[i][i+d] \leftarrow max(M[i][i+d],$
                $M[i][j] * M[j+1][i+d],$
                $M[i][j] * m[j+1][i+d],$
                $m[i][j] * M[j+1][i+d],$
                $m[i][j] * m[j+1][i+d]$
                $)$
                $m[i][i+d] \leftarrow min(m[i][i+d],$
                $M[i][j] * M[j+1][i+d],$
                $M[i][j] * m[j+1][i+d],$
                $m[i][j] * M[j+1][i+d],$
                $m[i][j] * m[j+1][i+d]$
                $)$
            if $M[i][i+d]$ *is updated in the previous if-else block* **then**
                $split[i][i+d] \leftarrow j$
return $M[0][n]$

Note: We needed to store both Maximum $M[i][j]$ as well as Minimum $m[i][j]$ because x_i's can be positive as well as negative, and as the product of 2 negative numbers can also be positive, one needs to store both the maximum possible value as well as the minimum possible values. If x_i's were constrained to only be positive, then one can just simply store the Maximum $M[i][j]$ and update it accordingly.

Time complexity: $O(n^3)$ as triple nested loops are used.

Reconstruction: Using $split[0][n]$, we can reconstruct the parenthesization output via traceback.
Recurrence relation explanation diagram:



- Written "I do not know how to approach this problem" - 0.6 points
-
  - Correct base case for the recurrence - 0.25 points
  - Correct recurrence relation - 1.25 points
  - Brief justification of the recurrence relation - 0.5 points
  - Mentioning the correct order of filling the DP table - 0.25 points
  - Outputting the optimal parenthesization (not just the optimal value) - 0.25 points
  - Brief justification of the time complexity - 0.5 points
  - **Note:** 0.5 points will be deducted if the solution is wrong in case some of x_i's are negative

4. Consider the following *interval stabbing* problem: We are given a set S of n intervals, where the i^{th} interval is denoted by $[\ell_i, r_i]$ and ℓ_i and r_i are integers. The goal is to determine if, for a given integer k, there exists a set of k integers $\{x_1, x_2, \ldots, x_k\}$ such that each interval in S contains at least one of the x_i's. Design a polynomial-time dynamic programming algorithm for this problem.

Sort all the intervals in non-decreasing order of their right end-points.
Let $P[i]$ denote the tuple (stab point x_i that stabs the i^{th} interval, total stab points made till i^{th} interval).
Base case: $P[1] = (r_1, 1)$ i.e. r_1 has stabbed the first interval.

Recurrence relation: We describe the following recurrence relation to find the minimum number of integers needed to stab all the intervals:

a) If $l_i \leqslant P[i-1][1]$ i.e. the i^{th} interval is stabbed by the $(i-1)^{th}$ interval's stab point: $P[i] = P[i-1]$

b) If $l_i > P[i-1][1]$ i.e. the i^{th} interval starts after the $(i-1)^{th}$ interval's stab point: $P[i] = (r_i, P[i-1][2] + 1)$

Checking if solution size $\leqslant k$: Check whether total stabs made till the last interval is limited upto k i.e. return $P[n][2] \leqslant k$.

Time complexity: Each element of the 1D DP array takes $O(1)$ time. Hence, $O(n)$. But the initial sorting takes $O(n \log n)$.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Correct base case for the recurrence - 0.25 points
  - Correct recurrence relation - 1.5 points
  - Brief justification of the recurrence relation - 0.5 points
  - Mentioning the correct order of filling the DP table - 0.25 points
  - Brief justification of the time complexity - 0.5 points

5. In this exercise, we will think about a voting problem involving the election of a committee. Let $C = \{c_1, c_2, \ldots, c_m\}$ denote the set of m candidates, and $V = \{v_1, v_2, \ldots, v_n\}$ denote the set of n voters. Each voter v_i has a strict and complete ranking R_i over the candidates in C.

   Consider the following voting rule f for selecting a committee of k candidates: Given any k-sized committee of candidates $W \subseteq C$, the score that voter v_i assigns to the committee W is equal to $m - j$ if voter v_i's *favorite* candidate in W is ranked at j^{th} position in its ranking R_i. The score of the committee W is the *minimum* of the scores it receives from all voters. The voting rule returns a committee with the highest score.

   For example, suppose there are three voters v_1, v_2, v_3 and five candidates $c_1, \ldots, c_5$. The rankings of the voters are:

   $$v_1 : c_1 \succ c_2 \succ c_3 \succ c_4 \succ c_5,$$

   $$v_2 : c_4 \succ c_2 \succ c_1 \succ c_5 \succ c_3,$$

   $$v_3 : c_5 \succ c_4 \succ c_3 \succ c_2 \succ c_1.$$

   Then, the committee $\{c_2, c_5\}$ gets a score of 3 from voter v_1 because its favorite candidate in the committee, namely c_2, is ranked second. Likewise, the committee gets scores of 3 and 4 from voters v_2 and v_3, respectively, resulting in an overall score of $\min\{3, 3, 4\} = 3$.

   We will assume that voter's preferences have the *single-peaked* property, defined as follows (see Figure 1): Consider an axis (or an ordering) σ of the candidates (not to be confused with the voters' preferences). Let R be any ranking of the candidates, and let $\texttt{top}(R)$ denote the top-ranked candidate in R. We say that R is single-peaked with respect to the axis σ if, for every pair of candidates c and c' that lie on the same side of $\texttt{top}(R)$ along the axis, if c is closer to $\texttt{top}(R)$, then c is preferred over c' in R. In other words, among the candidates on the same side of the "peak", the ones closer to the peak are preferred over those further away.

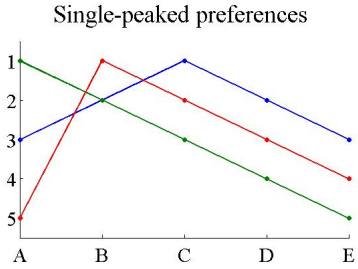


Figure 1: Example of single-peaked preferences. The axis is (A, B, C, D, E). The red voter corresponds to the ranking $B \succ C \succ D \succ E \succ A$. Thus, the "peak" of the red ranking is B. For any pair of candidates on the same side of the peak (say, C and D), the one closer to the peak is preferred over one that is further away. Image source: https://en.wikipedia.org/wiki/Single_peaked_preferences.

Show that when the rankings $R_1, \ldots, R_n$ are single-peaked (with respect to a known axis σ), the output of the voting rule f can be computed in polynomial time.

Hint 1: You may use the following equivalent definition of single-peaked preferences: For any k, the set of top-k candidates in each ranking constitutes a connected segment with respect to the axis. For example, consider the red voter's ranking in Figure 1. Observe that the top-k prefixes are $\{B\}$, $\{B, C\}$, $\{B, C, D\}$, $\{B, C, D, E\}$ and $\{B, C, D, E, A\}$.

Hint 2: You may find it useful to refer to the interval stabbing problem.

Informal Idea: To solve this problem, we can use binary search on the possible score values to find the maximum score that can be achieved by a committee of k candidates. Here's the approach in brief:

a) **Binary Search on Score**: We assume a potential score s and check if it's achievable for a committee of k candidates. The goal is to find the highest score s for which a valid committee exists.

b) **Define Intervals for Each Voter**: For each voter v_i, consider the candidates that v_i would rate with at least a score s. Since the preferences are single-peaked, it can be seen that this set of candidates will form a contiguous interval along the axis σ representing candidates. Let's call this interval $[l_i, r_i]$. As a corollary, this interval is defined such that each candidate within it is at most j-th in the voter's ranking, where $m - j \geqslant s$.

c) **Interval Stabbing Problem**: The problem now reduces to finding k candidates that collectively "stab" the intervals $[l_i, r_i]$ over all i. In other words, each interval $([l_i, r_i])$ must contain at least one of these k candidates. If such a selection of candidates exists, then a score of s is achievable.

Time Complexity: This algorithm performs $\log n$ iterations due to binary search on scores. In each iteration:

- It takes $O(n \log m)$ to calculate the intervals for all voters. This is because, for each voter, one can perform binary search to find the interval leftmost and rightmost points of the interval (**Try it Yourself**).
- It takes $O(n)$ time to solve the interval stabbing problem.

Therefore, the overall complexity is $O(n \log m \log n)$, which is polynomial.

6. ($\star$) How will your answer to the previous problem change if, instead of the minimum, the score of committee W is the *sum* of the scores it receives from all voters?

   Hint: Think about the "marginal" gain achieved by including a candidate in the committee relative to a voter's peak. The subproblems can be framed in terms of the size of the committee and the rightmost candidate in the committee.

   Informal Idea: Let $dp[i][j]$ denote the maximum score that can be achieved by forming a committee of j candidates, where the rightmost candidate in this committee is c_i (according to the fixed axis σ).
   For each voter v, determine the "marginal gain" for adding candidate c_i to a partially-formed committee. The score contribution from v depends on how close c_i is to their peak candidate in v's ranking. Specifically, for each candidate c_i, compute the gain it contributes when included in a committee of j candidates, given its position in relation to the voters' rankings.

   Recurrence Relation: The transition can be framed as:

$$dp[i][j] = \max_{1 \leqslant t < i} \left(dp[t][j-1] + \sum_{v \in V} (\text{score}_v(c_i) - \text{score}_v(c_t))_+ \right)$$

   where $x_+ = \max(x, 0)$ and $\text{score}_v(c_i)$ is the score given to candidate c_i by voter v, and c_t is a candidate already included in the committee.
   This formulation captures the additional value that candidate c_i brings relative to an already included candidate c_t for each voter.

   Base Case: Initialize $dp[i][1]$ as the score from selecting only candidate c_i, summing the scores from all voters for that single candidate.
   Thus the maximum achievable score would be $\max(dp[k][i])$ over all candidates $c_i \in C$.

   Proof of Correctness

   Inductive Hypothesis: The dynamic programming algorithm correctly computes $dp[i][j]$, which represents the maximum score obtainable by forming a committee of size j with the rightmost candidate being c_i.
   Base Case: For $j = 1$ (committees of size 1):

$$dp[i][1] = \sum_{v \in V} \text{score}_v(c_i)$$

This is correct because the score for a single candidate is simply the sum of scores assigned to that candidate by all voters. Therefore, $dp[i][1]$ accurately reflects the maximum score for each individual candidate.

Inductive Step: Assume that the algorithm correctly computes $\text{DP}[t][j-1]$ for all $t < i$ and for committee size $j-1$.
The recurrence relation computes $dp[i][j]$ by taking the best achievable score from a committee of size $j-1$ ending with some c_t, then considering the score improvement by adding c_i to this committee.
We analyze the improvement in the score based on the position of each voter's peak candidate relative to c_t and c_i, where c_t is the rightmost candidate of the committee of size $j-1$, and c_i is the candidate added to reach size j.

a) **Voter's Peak to the Left of c_t:**
   If a voter's peak is to the left of c_t, then the voter will prefer a candidate to left of c_t (including c_t) over c_i. Hence, the inclusion of c_i does not affect the score relative to this voter.

b) **Voter's Peak Between c_t and c_i:**
   i. **If $\textbf{score}_v(c_i) > \textbf{score}_v(c_t)$:** In this case, c_i is closer to the voter's peak than c_t, so the voter would prefer c_i over c_t. This leads to a gain of $\text{score}_v(c_i) - \text{score}_v(c_t)$ as voter v would prefer c_i over c_t.
   ii. **If $\textbf{score}_v(c_i) \leqslant \textbf{score}_v(c_t)$:** Here, the voter prefers c_t over c_i, so adding c_i does not increase the score relative to this voter.

c) **Voter's Peak to the Right of c_i:**
   If a voter's peak lies to the right of c_i, the voter prefers candidates closer to the peak, so they would prefer c_i over c_t. Thus, including c_i provides a score gain of $\text{score}_v(c_i) - \text{score}_v(c_t)$.

The recurrence computes $dp[i][j]$ by taking the best achievable score from a committee of size $j-1$ ending with some c_t, then considering the score improvement by adding c_i to this committee according to the above cases.

Time Complexity: Dimension of dp table is $m \times k$ and to calculate each dp value, we require $O(mn)$ time. Thus the total time complexity is $O(m^2kn)$.

7. ($\star$) Uh oh, another voting problem.

   This time, we will discuss a one-dimensional model of voting with a *continuum* of voters. Specifically, each point in the interval $[0, 1]$ denotes a *voter*. Additionally, there is a finite set of *candidates* $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ such that for every i, we have $c_i \in [0, 1]$. You may assume that c_i's are distinct. For any voter $v \in [0, 1]$, its *favorite* candidate, denoted by $\texttt{top}(v)$, is the candidate that is closest to it. That is, $\texttt{top}(v) \coloneqq \arg\min_{c \in \mathcal{C}} |v - c|$.

Unfortunately, in our problem, voters cannot vote directly for their favorite candidate; instead, they must delegate their vote to a *proxy*, who then votes for its own favorite candidate.[1] Specifically, a proxy is represented by a point in $[0, 1]$, and, in general, there can be multiple proxies. We will denote the set of proxies by $\mathcal{P} = \{p_1, p_2, \ldots, p_k\}$. For any voter v, its *closest proxy* is given by $p_v := \arg\min_{p \in \mathcal{P}} |v - p|$. Each voter v delegates its vote to its closest proxy p_v, who then votes for its own favorite candidate given by $\texttt{top}(p_v) := \arg\min_{c \in \mathcal{C}} |p^v - c|$. Note that due to the delegation process, a voter's vote may get *distorted*, and the voter may end up voting for a non-favorite candidate, i.e., $\texttt{top}(p_v) \neq \texttt{top}(v)$ (see Figure 2). Our goal is to find an arrangement of the proxies so that this distortion is minimized.

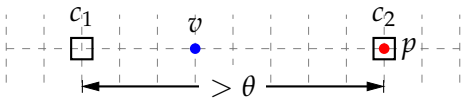


Figure 2: Failure of θ-representation. Voter v's favorite candidate is $\texttt{top}(v) = c_1$. However, its closest proxy is p, whose favorite candidate is $\texttt{top}(p_v) = c_2$. Since $|c_1 - c_2| > \theta$, the vote is significantly distorted.

Formally, given a threshold $\theta \in [0, 1]$, an arrangement of proxies $\mathcal{P}$ is said to be *θ-representative* if, for every voter $v \in [0, 1]$, we have $|\texttt{top}(p_v) - \texttt{top}(v)| \leqslant \theta$. Your task is to design a polynomial-time algorithm that, given as input the positions of the m candidates, a budget of k proxies (where $k \leqslant m$), and a distortion threshold θ, determines whether there exists a θ-representative proxy arrangement. If the answer is YES, your algorithm should also return the corresponding proxy arrangement.

For this problem, you can assume that the proxies are chosen from among the candidates, i.e., $\mathcal{P} \subseteq \mathcal{C}$. Thus, in particular, if proxy p lies at the same position as candidate c (as in Figure 2), then p obviously votes for candidate c.[2]

Informal Idea: We break up the problem into two parts. In the first part we figure out if we choose two proxies c_i and c_j, $j > i$ such that no candidate c_l, $i < l < j$ is chosen as a proxy, then can we satisfy θ-representation for every voter between c_i and c_j. Next we use this sub-solution to construct our solution in which we choose the proxies. This can be done by a dp on the first i candidates in which c_i is chosen as a proxy and θ-representation is satisfied for all voters before c_i.

Notation: $(x \ ? \ y : \ z)$ denotes the if-then-else operator, i.e., if x holds, then return y else return z.

DPs:

- $X_{i,j}$: It is 1 if after choosing c_i and c_j as proxies and no proxies in between, every

[1] See https://en.wikipedia.org/wiki/Proxy_voting and https://en.wikipedia.org/wiki/Liquid_democracy.

[2] If you can solve this problem *without* assuming that proxies are chosen from among the candidates, please get in touch with Rohit.

voter between c_i and c_j is θ-represented, otherwise 0. More formally

$$X_{i,j} \equiv \forall_{v \in [c_i, c_j]} |\texttt{top}(p_v) - \texttt{top}(v)| \leqslant \theta$$

- S_i: It is 1 if by only choosing c_i as a proxy we satisfy θ-representation for all voters $v \leqslant c_i$, otherwise 0.
- E_j: It is 1 if by only choosing c_j as a proxy we satisfy θ-representation for all voters $v \geqslant c_j$, otherwise 0.
- Y_i: It is the minimum number of proxies that need to be chosen from $\{c_1, \ldots, c_{i-1}\}$, such that along with c_i we get a set of proxies which satisfy θ-representation for all voters $v \leqslant c_i$

Computing X, S and E:
As it is impossible to compute θ-representation for every real point in $[0, 1]$, we only consider some special points for which if θ-representation is satisfied then it would be satisfied for every point. Lets denote this set of points as V, which is defined as:

$$V = \{c_{i+1} - c_i \mid i \in [m-1]\}$$

Now we can define our algorithms for computing S, E and X.

ALGORITHM 2: Computing S, E and X

Input: The set of candidates $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$
Output: The DPs S, E and X

```
for i ← 1 to m do
    S_i ← 1
    V' ← V ∩ [0, c_i]
    for v ∈ V' do
        if |c_i − top(v)| > θ then
            S_i ← 0
    E_i ← 1
    V' ← V ∩ [c_i, 1]
    for v ∈ V' do
        if |c_i − top(v)| > θ then
            E_i ← 0
    for j ← i + 1 to m do
        V' ← V ∩ [c_i, c_j]
        X_{i,j}
        for v ∈ V' do
            if |c_i − v| < |c_j − v| then
                c' ← c_i
            else
                c' ← c_j
            if |c' − top(v)| > θ then
                X_{i,j} ← 0
return {S, E, X}
```

Now that we have computed the auxiliary DPs, we can compute Y.

Base case:

$$Y_i = S_i \; ? \; 0 : \infty$$

Recurrence Case:

$$Y_i = \min(Y_i, \min_{j<i}(X_{i,j} \; ? \; Y_j + 1 : 0))$$

Order of iteration: We need to compute Y_i from $i = 1 \rightarrow m$

Finding the solution:
After Y_i is computed, we can find the minimum number of proxies required:

$$Ans = \min_{i \in [m]}(E_i \; ? \; Y_i + 1 : \infty)$$

If $Ans \leqslant k$ then we have a valid arrangement of proxies. To actually get this set of proxies we can find the value of i for which $E_i = 1$ and Y_i takes minimum value. c_i will

be one proxy. Then we can find Y_j such that $Y_j = Y_i - 1$ and $X_{i,j} = 1$, and we would get c_j as another proxy. We repeat this until we get all proxies.