PROJECT

Advanced Algorithm and Programming

Teacher : Mr Khosravian

Group : Maryline CHEN
Yves TRAN

Master 1 – Informatique, Décision, Données
Université Paris-Dauphine

I. Discrete Unit Disk Cover Problem

A. One dimensional plane

1. Propose a greedy algorithm for DUDC problem in a 1-dimensional plane and prove your algorithm returns the optimal solution of DUDC in 1-dimensional plane.

Input : a_i, b_i and x_j can be floats or integers

- Array of same size intervals Q = $[(a_1,b_1),(a_2,b_2),...(a_n,b_n)]$: the bounds of the interval are included
- Array of points P = $[(x_1,0),(x_2,0),...,(x_k,0))]$

Output : A set I ⊆ Q of minimum size that covers all points in P.

Pseudo-code :

algorithm dudcONE **is**

```
1.  sort P in ascending order
2.  sort Q in ascending order (= sort the intervals by their lower bound)
3.  counter <-- 0
4.  I <-- [ ]
5.  j <-- 0
6.  for j=1 to k{
7.      if I ≠ ∅ then
8.          if P[j] in the last interval added in I then
9.              counter <-- counter + 1
10.     else if j >= size(Q) then
11.         return []
12.     else
13.         tmp <-- counter
14.         for i=j to size(Q) {
15.             if P[j] in the interval Q[i] then
16.                 if (P[j] is in the interval Q[i]) and ((not in the interval Q[i+1]) or (Q[i] is the last interval in Q)) then
17.                     counter <-- counter + 1
18.                     add Q[i] to I
19.                     break
20.             j <-- j + 1
21.         if tmp = counter then
22.                 return []
23.         }
24. }
25. if counter = size(P) then
26.     return I
27. else
28.     return []
```

Optimality proof :

Recall : we are looking for a subset $Q^* \subseteq Q$ of **minimum size** that covers all points in P.

Let I be the solution of our algorithm in 1-dimensional plane.
Let suppose J is the optimal solution and $J \neq I$.

⚠ **Intervals in Q are same size** ⚠

We can distinguish **3 cases** :

- **Case size(J) = size(I)** : Both are optimal since we are looking for a subset of minimum size that covers all points in P. So I is optimal.
- **Case size(J) > size(I)** : Then J is not optimal
- **Case size(J) < size(I)** : we have two subcases : $J \not\subset I$ and $J \subset I$

We will only prove one of the two since they are quite equivalent since :

⇨ **if $J \not\subset I$**, I covers all the points using too more intervals than necessary and contains some different intervals than the J but it doesn't necessarily mean that they're not as "optimal" intervals as those in J's. (Sometimes, we can have different set of optimal solution).
So, I can contains "optimal" intervals and "useless" intervals.

⇨ **if $J \subset I$**, I covers all the points using too more intervals than necessary : the set $I \backslash J$ is useless. This case is quite the same as the previous case since it includes this one, that's why we will only study $J \not\subset I$.

IF $J \not\subset I$:

a) case : ∃ **m** such that the interval **I[m] covers Ø point**

Not possible since an interval I[i] is added to I if a point belongs to it and doesn't belong to I[i+1]. So the minimum requirement to be added is to cover at least one point.
If there is no interval in I that contains our point, we prefer a 'higher/bigger' interval in terms of lower bound that contains our point because a higher may contains some points we haven't seen yet. **So all intervals in I covers at least one point $p \in P$.**

b) case : ∃ **m'** such that the interval **I[m'] covers point(s) that is already covered** and it was not necessarily to take it

Not possible since before adding any interval, we check first if it is already covered by an interval stored in I.

Conclusion :
Any other interval different from those in J, are as optimal as those in J.
Moreover, it is not possible for I to have 'useless' intervals according a) and b) so I is optimal.
In this way, it is not possible that size(J) < size(I) since I is optimal, so there is no less than size(I) intervals that covers all the points : J is not optimal.

NB :
Our algorithm dudcONE returns always a list :
-> can empty if there is no solution to our problem, or the given set Q is empty.
-> if there is a solution, returns the optimal solution : this solution always covers all the point, since before returning any set I, it checks if the number of covered points is equal to size(P).

2. Implement your greedy algorithm for a 1-dimensional plane.

   Answer : ***cf code***

B. Two dimensional plane

3. For the 2-dimensional plane, prove DUDC problem is NP-hard.

NP-hardness proof

In order to prove the NP-hardness of DUDC, we can reduce it to the Minimum Steiner Tree problem (MST).

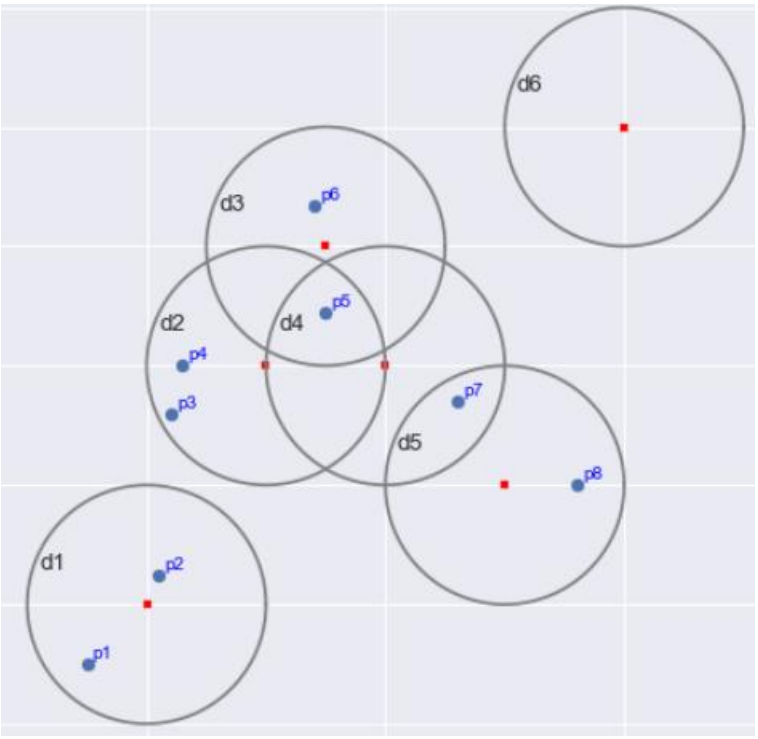
Let ***P*** a set of points and ***Q*** a set of disks in the DUDC problem. Let ***k*** be an integer.
The DUDC problem can be formulate as follow : is there a subset $Q^* \subseteq Q$ such that $|Q^*| \leq k$?

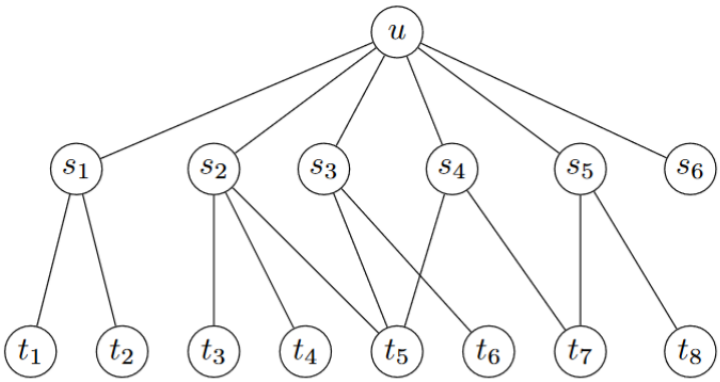
Graph construction

Let ***G=(V, E)*** be the final graph. We call ***T*** the terminal nodes and ***S*** the Steiner nodes.

The idea of the construction is the following :

- For each disk $d \in Q$, create a Steiner node s_d and add it to ***S***, the set of Steiner nodes.
- For each points $p \in P$, create a terminal node t_p and add it to ***T***, the set of terminal nodes.
- Create a node $u \in T$ and add an edge between ***u*** and s_d , $\forall\ s_d \in S$.
- If a point $p \in P$ belongs to a disk $d \in Q$, then add an edge between t_p and s_d , their corresponding node in the graph G.





Reduction proof

Let's show that **there is a solution for DUDC of size** *k* iff **there is a solution of size** k+ |*T*| **for MST** (size = number of edges).

Sufficiency : Suppose there is a minimum Steiner tree *R** in G. All Steiner nodes must be covered. One can show that the covered nodes in ***S*** in MST corresponds to a set of disks *Q** in DUDC. Indeed, let a point ***p*** $\in$ ***P*** from DUDC and its corresponding node $t_p \in T$ in MST. The parent of t_p in *R** are some Steiner nodes {s_1, s_2, …, s_l}. Since t_p is linked to s_i iff the corresponding disk of s_i contains ***p***, taking the corresponding disk of s_i in *Q** will lead us to a solution of size *k* for DUDC (since there are only *k* edges from u to ***S*** which are in *R**).

Necessity : Suppose we have a solution *Q** for DUDC. We can build a Steiner tree *R* by taking the edges from *u* to the Steiner nodes that correspond to a disk in *Q** and the edges from these Steiner nodes to all terminal nodes (if a terminal node is adjacent to two or more of these Steiner nodes, we take only one of its incident edges). All terminal nodes must be covered since all points of *P* are covered in DUDC. Moreover, *R* does not contain any cycle, therefore *R* is a Steiner tree of size ***k+|T|.***

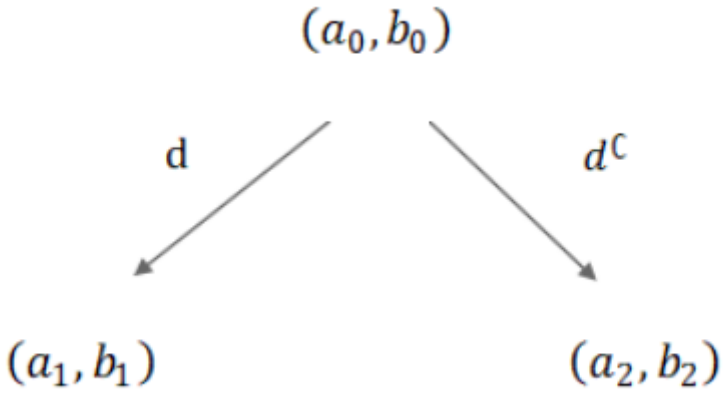
In conclusion, there is a solution for DUDC of size ***k*** ⇔ there is a solution of size ***k+|T|*** for MST. Therefore, DUDC is NP-hard.

4. Propose a Branch-and-bound algorithm for the problem in 2-D, and clearly explain what is your bound procedure (Heuristic function)?

In order to use tackle the problem with a Branch-and-bound algorithm, we have chosen the bound of a node to be the length of the current solution (denoted as **a**) and considered the number of covered points (denoted as **b**). Therefore, a node of the Branch-and-bound tree can be denoted as **(a, b).**

The idea of the Branch-and-bound procedure is the following : at each step, the set of uncovered points and unused disks are considered. One disk is selected from the set of unused disks to be reviewed. Thus, two child nodes can be created, one that contains the selected disk in its solution and one that does not. The selected disk is seen as used in the child nodes. The process is repeated for each node of the Branch and Bound tree as long as their solution is feasible. **A candidate solution is no more considered as feasible if there is a point which cannot be covered** (either by the node's solution or by the remaining unused disks) or a complete solution with a smaller size has been found.

On the example below, from node (a_0, b_0) is created two child node (a_1, b_1) and (a_2, b_2). (a_1, b_1) contains a solution set with disk d while (a_2, b_2) contains a solution set without d.



Since each node can be seen as a subtree, **we will only talk about trees instead of nodes.**
To simplify the notation and structure of the algorithm, we define a *Tree* structure as follow :

Tree <- {

- sol : set of disks which represents the current solution
- $\tilde{P}$: $set\ of\ remaining\ points\ to\ cover$
- $\tilde{Q}$: set of remaining disks to consider
- bound : heuristic value
- Ch : set of tree which represent children of the current tree

}

If T is a tree, we denote T.Ch the children of T, T.bound the bound of T, etc.

Input : P, Q

- Array of points P (points are represented by their coordinates) $= \{p_1, p_2, \dots, p_n\}$
- Array of unit disks Q (disks are represented by their center) $= \{q_1, q_2 \dots, q_m\}$

Output : An array of disk Q'

Choosing a node : The chosen node to develop is the one that have the lowest ratio $\frac{solution\_size}{number\_of\_covered\_points}$. Indeed, we want the optimal solution to be found quickly and it must have a minimum solution size covering a maximal set of points.

Sorting Q : It is possible to sort Q before starting the algorithm by number of covered points in descending order to reduce the tree's depth. The intuition is that a disk which covers a lot of points is likely to be in the final solution. Still, it is not always true, that's why sorting is let to be optional.

Pseudo-code :

algorithm dudcTWO **is**

1. sort Q in decreasing order of number of covered points (optional)
2. root <-- new Tree
3. to_visit <-- [root]
4. Q' = [ ]
5. **while** to_visit is not empty {
6. tree_to_develop <-- tree with lowest ratio $\frac{solution\_size}{number\_of\_covered\_points}$ in to_visit
7. remove tree_to_develop from to_visit
8. children = **split**(tree_to_develop)
9. **for** (child in children) {
10. Add child to tree_to_develop.Ch
11. **if** child is realizable but not a complete solution **then**
12. to_visit.append(child)
13. **else if** child is complete solution and len(child's solution) < len(Q') **then**
14. Q' <-- child's solution
15. }
16. }
17. **return** Q'

Input : Tree T

Output : Two children T1, T2

Pseudo-code :

algorithm split **is**

1. if no points nor disks left in T then
2. return None
3. T1 <-- new Tree
4. T1. $\tilde{P}$ = T.$\tilde{P}$ – {points covered by T.$\tilde{Q}$[0]}
5. T1. $\tilde{Q}$ = T.$\tilde{Q}$ – {T.$\tilde{Q}$[0]}
6. T1.sol = T.sol $\cup$ T. $\tilde{Q}[0]$
7. T1.bound = len(T1.sol)
8. T1.Ch = [ ]
9. T2 <-- new Tree
10. T2. $\tilde{P}$ = T.$\tilde{P}$
11. T2. $\tilde{Q}$ = T.$\tilde{Q}$ – {T.$\tilde{Q}$[0]}
12. T2.sol = T.sol
13. T2.bound = len(T2.sol)
14. T2.Ch = [ ]
15. **return** array([T1, T2])

5. Implement your Branch-and-bound algorithm for a 2-dimensional plane.

   Answer : ***cf code***

II. Upper Envelope of Some Linear Functions

A. Divide and Conquer

1. Propose a Divide-and-Conquer algorithm for the Upper Envelope Problem

Input : m_i, b_i and x_j can be floats or integers $\forall\, i \in [1, n]$, $\forall\, j \in [1, k]$

- Array of function's coordinates **Y** = [y_1, …, y_n] where $y_i = (m_i, b_i)$ $\forall\, i \in [1, n]$
- Array **x** = [x_1, x_2, x_3, ... , x_k]

Output : A list **uE** of sublists, where each sublist contains 2 tuples :

- each sublist of uE corresponds to a fragment of the upper envelope
- first tuple : coordinate of the function
- second tuple : index of x, where the function in first tuple is >= the other functions in Y

Idea explanation : we can see the upper envelope as the union of functions over intervals segment of x.

Example : Y = [(m_1,b_1) , (m_2,b_2)] , x = [x_0 , ... , x_{99}]
uE = [[(m_2, b_2) , (0, 45)] , [(m_1, b_1) , (46, 99)]]

The first sublist : [(m_2, b_2) , (0, 45)] means the second function y_2 is >= to y_1 on [x0,x45]
The second sublist : [(m1, b1) , (46, 99)] means the first function y1 is >= to y2 on [x46,x99]

Pseudo-code :

algorithm findUE_DC **is**

1. **if** size(Y) = 1 **then**
2. **return** Y[0]
3. **else**
4. y_1 <-- Y[y_1, ... , $y_{\lfloor \frac{n}{2} \rfloor}$]
5. y_2 <-- Y[$y_{\lceil \frac{n}{2} \rceil}$, ... , y_n]
6. y_1 <-- findUE_DC(x, y_1)
7. y_2 <-- findUE_DC(x, y_2)
8. **return** conquer(x, y_1 , y_2)

Input : x_j can be floats or integers $\forall\ j \in [1, k]$

- Array **x** = [x_1, x_2, x_3, ... , x_k]
- f, g : single coordinate of function **or** list of sublists, where each sublist contains 2 tuples

Output : A list **uE** of sublists, where each sublist contains 2 tuples

Pseudo-code :

algorithm conquer **is**

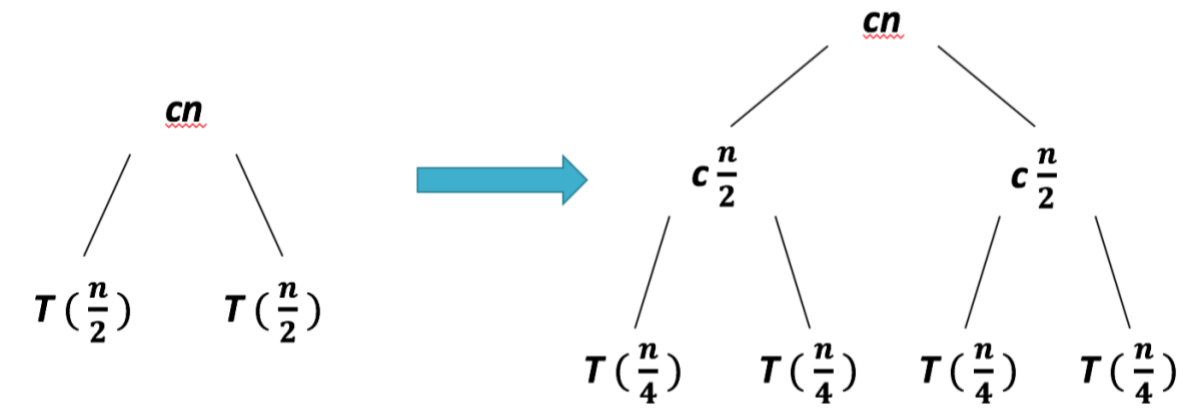
```
1.  upperE <-- [ ]
2.  calculate f over x
3.  calculate g over x
4.  h <-- sign of f - g
5.  idx <-- index of points extremum points
6.  if size(idx) = 0 then
7.      if ( '<' in h ) then
8.          add coordinates of corresponding function in g over x to upperE
9.      else
10.         add coordinates of corresponding function in f over x to upperE
11. else
12.     for i=1 to size(idx)+1{
13.         if i = 0 then
14.             start <-- i, end <-- idx[i]
15.         else if i = size(idx) then
16.             start <-- idx[i-1]+1, end <-- size(h)
17.         else
18.             start <-- idx[i-1]+1, end <-- idx[i]
19.         if ( '<' in h[start, ..., end] ) then
20.             add coordinates of corresponding function in g over x to upperE
21.         else
22.             add coordinates of corresponding function in f over x to upperE
23.     }
24. return upperE
```

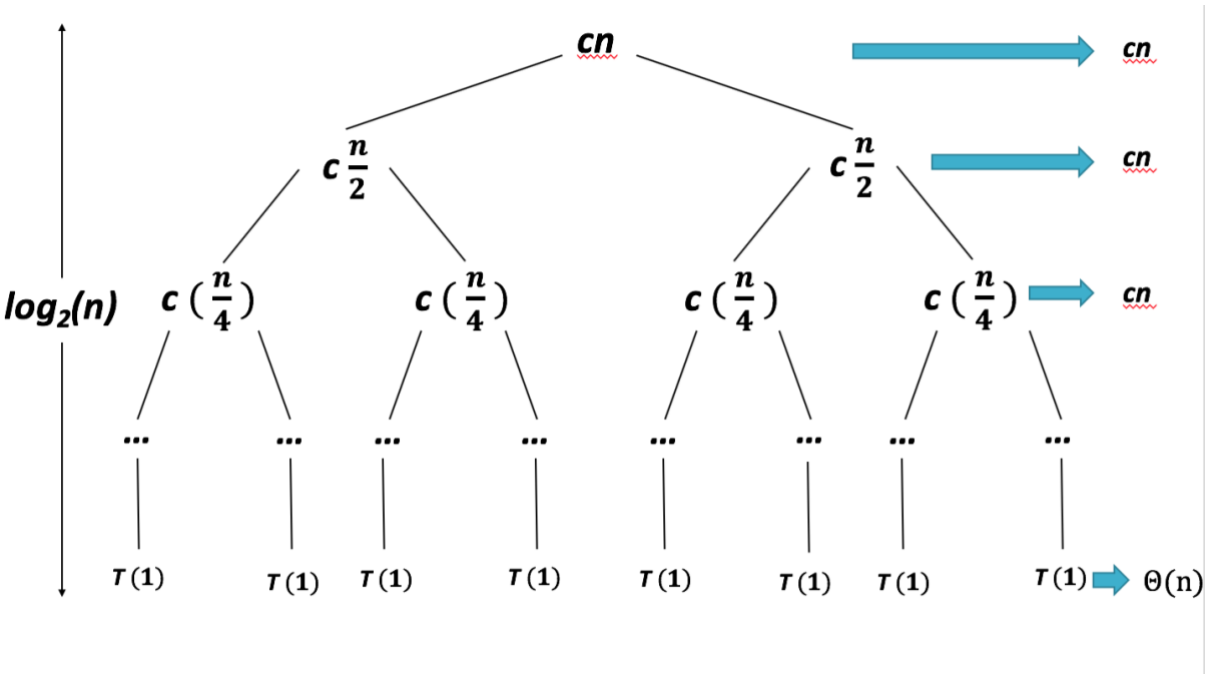
2. Compute the time complexity of the D&C algorithm by recursion “tree method”.

Let n be the number of functions. The algorithm findUE_DC seems to be in $O(n\log_2(n))$.
Let’s prove it.

First of all, in our function *divide(Y)*, we divide the subset Y by 2 until having only one set. We can estimate the complexity of this function by $\log_2(n)$.
Then, in our function conquer(x,f ,g), we iterate on the same time f and g : once, to substract g to f, and a second one in getFunctionOn(f, start, end). So conquer(x, f, g) is in $\Theta(2*n)$. In other words, in O(n).

We can summarize it by a recursive tree :





So, we have for findUE_DC :

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

By recursion tree method, we get :

$$T(n) = cn + cn + cn + \dots + \Theta(n)$$

$$= \sum_{i=0}^{log_2(n-1)} (\frac{2}{2})^i \, cn + \Theta(n) = \sum_{i=0}^{log_2(n-1)} cn + \Theta(n)$$

$$= log_2(n-1) + 1 \ * cn + \ \Theta(n) = O(n\log_2(n))$$

Thus, findUE_Dc is in $O(n\log_2(n))$

3. Implement the proposed D&C algorithm.

   Answer : ***cf code***

4. Propose a Dynamic Programming algorithm for the Upper Envelope Problem.

Input : m_i, b_i and x_j can be floats or integers $\forall$ i $\in$ [1, n], $\forall$ j $\in$ [1, k]

- Array of function's coordinates **Y** = [y_1, …, y_n] where y_i = (m_i, b_i) $\forall$ i $\in$ [1, n]
- Array **x** = [x_1, x_2, x_3, ... , x_k]

Output : A list memo[e_1, ..., e_n] where :

- e_1 : upper envelope of (m_1,b_1)
- e_i : upper envelope between (j_{i-1} , (m_i , b_i)) $\forall$ i $\in$ [2, n]

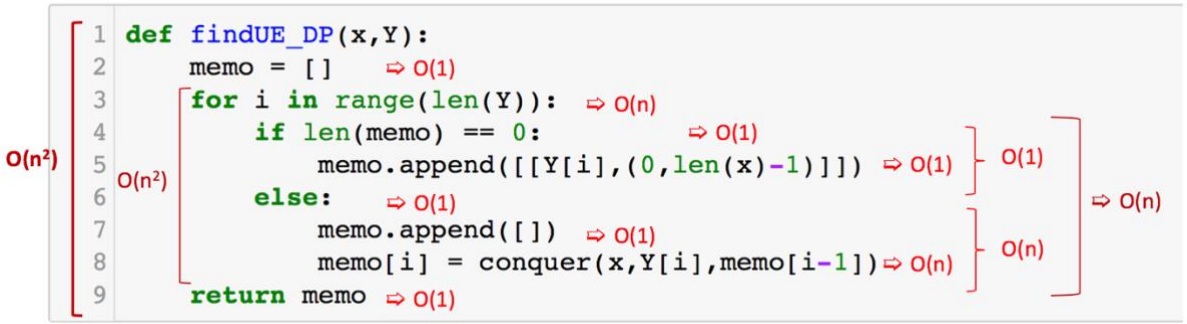
Pseudo-code :

algorithm findUE_DP **is**

```
1. memo <-- [ ]
2. for i=1 to size(Y) {
3.     if size(memo) = 0 then
4.         add [ [ Y[i] , (1, size(x)) ] ] to memo
5.     else
6.         add [ ] to memo
7.         memo[i] <-- conquer(x, Y[i], memo[i-1])
8. }
9. return memo
```

5. Compute the time complexity of the DP algorithm.

The time complexity of the Dynamic Programming algorithm is $O(n^2)$. Let's analyze line by line the time complexity.



In our case, the Dynamic Programming seems using more time than the Divide and Conquer algorithm :

$$O_{DC}(n\log_2(n)) < O_{DP}(n^2) \quad \forall\ n \in \mathbb{N}$$

Indeed, when we measure the time used for each function, DP algorithm used a little bit more time than DC algorithm. The difference of time between those functions increases wih the size of the instance.

6. Implement the proposed DP algorithm.

   Answer : ***cf code***