# DES227
## Algorithm Design : Lecture 2-7 (Fundamental - Dynamic Programming)

**Fundamentals of Analysis (Lecture 2)**

- focused on 2 types of efficiency : Time & Space
- investigate the efficiency by looking at the input size of algorithm
- measuring run time by counting the basic operation is executed, basic operation is the most time-consuming operation in the algorithm's innermost loop.
- look at the order of growth of the algorithm
- Class of efficiency $1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$
- worst-case : runs the longest among all possible input of size n in that algorithm
- best-case : runs the fastest
- average-case : typical/random, make assumption about input

**Asymptotic Notation**

$O(g(n))$ : smaller or same order ; $f(n) \leq g(n)$ ex $n \in O(n^2)$

$\Omega(g(n))$ : larger or same order ; $f(n) \geq g(n)$ ex $n^3 \in \Omega(n^2)$

$\Theta(g(n))$ : same order of growth ; $f(n) = g(n)$ ex $n^2 \in \Theta(n^2)$

- $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- using limit to compare order of growth

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 ; & t(n) < g(n) \\ c ; & t(n) = g(n) \\ \infty ; & t(n) > g(n) \end{cases}$$ use l'Hopital's rule if necessary

**Basic Sum operation**

$$\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i \quad \Big| \quad \sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$$

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \Big| \quad \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \quad \Big| \quad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

**Analysis of nonrecursive**

determine input size → check basic operation → check basic ops & input whether it depends or not → set up sum of basic ops → find order.

**Analysis of recursive**

set up the recurrence relation of the basic operation to solve for the order of growth of an algorithm

Ex recursive function → recurrence relation

$F(n) = F(n-1) \times n$ → $M(n) = M(n-1) + 1$

$F(0) = 1$ $M(0) = 1$

* basic ops is multiplication (one per level)

$M(n) = M(n-i) + i$

$M(n) = M(n-n) + n$

$= 0 + n$

$= n$

Thus, $M(n) \in \Theta(n)$ ✱

## Brute Force (Lecture 3)

- basically means "just do it" / straight-forward approach to solve problem / simplest of all design

## Selection Sort

- scan list → select smallest → swap with the current position of the list

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

## Bubble Sort

- compare adjacent, if out of order, swap it just like "bubbling up" to the front

$$C(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

## Sequential Search

- compare element in list, each, with the key until match is found or hit the end of the list.
- efficiency = $\Theta(n)$ ; worst-case: last element = key or not found
  best-case: first element = key ($\Theta(1)$)

## Brute force Match String

- find the pattern in given text by compare the first character, if it is the same, check with the next character, if not check first with next in text. Stop when, hit the last text-pattern comparison or match is found
- aka. align the pattern, stop when n-m+1 round
- efficiency: text is n, pattern is m, $\Theta(nm)$, basic ops: comparison

## Exhaustive Search

brute-force of combinatorial problems

- generate each and every problem's domain → select the satisfy → find the desired one

### Traveling Salseman

- finding shortest tour of given city without going to the same city unless back to start
  aka finding Hamiltonian circuit of graph

### Knapsack Problem

- given n items, weigh of each item $w_i$, values of each item $v_i$ and capacity W
  find most valuable subset to fit in the knapsack
- list all combination, select the one combination that has most value within capacity

## Divide-and-Conquer (Lecture 4)

- divide into smaller version of same problem, solve smaller in recursively, combine smaller solution to get the original solution of original problem
- not necessarily efficient than brute-force
- Theorem $T(n) = aT(n/b) + f(n)$ ; divided to b instance, a of them is to solved, f(n) is function to spent in dividing & combining

$$T(n) \in \begin{cases} \Theta(n^d) & ; a < b^d \\ \Theta(n^d \log n) & ; a = b^d \\ \Theta(n^{\log_b a}) & ; a > b^d \end{cases}$$

that $f(n) \in \Theta(n^d)$
$a, b \geq 1$ ; $d \geq 0$
*also $O(n)$ and $\Omega(n)$ too.

## Mergesort

- divide array into 2 smaller, sort smaller recursively, merge smaller into single array
- merge by compare first element from each smaller array, put in smaller first, then loop
- $C_{worst}(n) \in \Theta(n \log n)$, $C_{average}(n) \in \Theta(n \log n)$

**Quicksort**

- divide input's element according to the position in the array, partitioning
- partition into less than, equal, greater than pivot element
- then rearrange lessthan, pivot, equal, greater than into the set

$C_{best}(n) \in \Theta(n \log_2 n)$, $C_{worst}(n) \in \Theta(n^2)$, $C_{avg}(n) \in \Theta(n \log_2 n)$

**Binary Search**

- efficient way to search element in SORTED array
- if it match, stop.
  if it less than, search the left of array, otherwise search on the right
- $C_{avg}(n) \in \Theta(\log n)$

**Graph Algorithm (Lecture 5)**

- based on idea of decrease-and-conquer
  - reduce problem instance into smaller one
  - solve smaller one
  - extend smaller solution to gain original solution

3 approaches
- decrease by a constant (reduce by 1)
- decrease by a constant factor (reduce more than 1)
- decrease by varies size (varies size decrease in each loop)

**Insertion Sort.**

- start with A[1], ending with A[n-1]
  A[i] is inserted in its appropriate position in the sorted group
- require 2 loops; 1 for get the next element, 1 to sort in sorted group

$C_{best}(n) \in \Theta(n)$, $C_{worst}(n) \in \Theta(n^2)$, $C_{avg}(n) \in \Theta(n^2)$

**Selection Problem**

- find k-th smallest element in given list of n element
- using list partition advantage (well, sorted this part)
  * S is position of pivot, k is order that we want.
    - S=k; pivot is solution
    - S<k; search on right of pivot
    - S>k; search on left of pivot

  * idea like Halfway quicksort

**Depth-first Search (DFS)**

- go as deep as possible first, hits deadend and go back 1 level, then continue until all vertices are visited
  (like pop print push)
- efficiency: stored graph as adjacency matrix : $\Theta(V+E)$
  adjacency list : $\Theta(V^2)$

**Breadth-first Search (BFS)**

- visit adjacent vertex, first adjacent, then 2 degrees, 3, 4, ...
- use queue (deq, print, enq)

**Topological Sort.**

- directed acyclic graph only (dag)
- delete vertex with no incoming edge first, then repeated by deleting then yield a solution
- or perform DFS then reverse it, yield solution of this sort.

## Greedy Algorithm (Lecture 6)

- used to solve optimization problem
- obtain optimal solution by making a sequence of choice and makes choice that always look best at the moment (locally greedy choice)
- solve in top-down manner

## Coin Changing Problems

- make change for amount N using least number of coins of denomination $d_1 > d_2 > \dots > d_m$
- greedy: select largest available

* remember that greedy algorithm is not always yield optimal solution.

## Kruskal Algorithm

- used to find minimum spanning tree of a graph
- MST of weighted connected graph is the smallest sum of tree in the graph
- Greedy: add an edge of minimum weight that don't make a cycle.

1) sort edges by its weight in ascending order
2) choose minimum edge and added to the tree, if edge makes a circle, don't select it (skip)
3) select until you have V-1 edges (V-1 round)

- another way is union-find algorithm
  - cycle is created, iff, 2 vertics belong to the same connected component.
  - if cycle created, find(u) = find(v)
    find(u) is returned subset containing u.
- time efficiency is $\Theta(|E| \log |E|)$

## Prim's Algorithm

- construct MST by extending its subtrees
- Greedy: add an edge with minimum weight that has one verter in the current tree and the other not in the current tree
- Stop after all vertices are in the minimum spanning tree.
- on each iteration, expand the tree on the greedy rule.
- efficiency is represented by the storage of graph
  - weighted matrix: $\Theta(|V|^2)$
  - adjacency list: $\Theta((|E|+|V|)\log|V|)$

## Continuous Knapsack Problem

- Greedy rule: select item in decreasing (max-to-min) order of ratio of value to weight ratio, take all if capacity not exceed. if exceed the weight, take what ever portion of object that fills the knapsack and stop the algorithm

## Dijkstra's Algorithm

- solves the single-source shortest-path with all edges are positive number
- Relax Edge by if current cumulative distance is greater than previous distance + that weight edge then change the distance to that value
  (if $v.d > u.d + w(u,v)$, then $v.d = u.d + w(u,v)$)

## Bellman-Ford Algorithm

- solves for single-source shortest path with negative-weight edge or positive
- ☆ If there's a negative-weight cycle, no solution exist.
- runs in each iteration, for maximum V-1 times. / relax the edge everytime there's a known path to that vertices.

## Dynamic Programming (lecture 7)

- used to solve the optimization problem
- it suggests solving each of smaller subproblem only once and recording the result in table form which can obtain a solution to the original problem

Idea : 1) Structure of optimal solution 2) Recursively define value of optimal solution
3) Compute the value of an optimal solution from bottom-up
4) Construct solution from computed info.

ex fibonacci sequence
: remember first and second previous element instead of recursive call function over and over again.

## Coin Changing Algorithm

- we want to make change for an amount N using least number of coins in denomination $d_1 < d_2 < \dots < d_m$

where in this algorithm

$C[i,j]$ = the minimum number of coins to make change for the amount $j$ using coin $d_1, d_2, \dots$ through $d_i$

where $C[i,0] = 0$
solution to the $C[i,j]$ is $\min\{C[i-1,j], 1 + C[i, j-d_i]\}$

## 0/1 knapsack Problem

$V[i,j]$ ; the most valueable subset of the fist $i$ items that fit in to weight $j$ capacity

goal to find $V[n,W]$

$$V[i,j] = \begin{cases} \max[V[i-1,j], v_i + V[i-1, j-w_i]] \text{ ; if } j-w_i \geq 0 \\ V[i-1,j] \text{ ; if } j-w_i < 0 \\ 0 \text{ ; if } i \geq 0 \text{ and } j = 0 \\ 0 \text{ ; if } j \geq 0 \text{ and } i = 0 \end{cases}$$

ex $W = 5$

| | | |
|---|---|---|
| item 1 : | weight = 2 | value = 12 |
| item 2 : | weight = 1 | value = 10 |
| item 3 : | weight = 3 | value = 20 |
| item 4 : | weight = 2 | value = 15 |

then : V[i,j] table

| i \ j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 12 | 12 | 12 | 12 |
| 2 | 0 | 10 | 12 | 22 | 22 | 22 |
| 3 | 0 | 10 | 12 | 22 | 30 | 32 |
| 4 | 0 | 10 | 15 | 25 | 30 | 37 |

comp. comp. comp. comp.

① V[i,0], V[0,j] = 0

② V[1,1] capacity = 1 using item 1 only
→ $j - w_i = 1 - 2 = -1$
↳ V[1,1] = V[0,1] = 0

V[2,1] capacity 2 using item 1~2
→ $j - w_i = 2 - 2 = 0$
↳ V[2,1] = max(V[1,1], 10 + V[1,0])
= max(0, 10)
= 10

∴ Capacity 5
↓
most valuable subset is {item 1, item 2, item 4}

V[3,4] ⇒ $j - w_3 = 4 - 3 = 1$
= max(V[2,4], 20 + V[2,1])
= max(15, 30) = 30

③ V[n,W] = solution = 37

④ traceback
V[4,5] > V[3,5] → item 4 is in the knapsack
→ capacity from 5 to 3 (5-2)
V[3,3] = V[2,3] → item 3 is not in the knapsack
→ move on to item 2 with capacity 3
V[2,3] > V[1,3] → item 2 is in the knapsack
→ capacity from 3 to 2 (3-1)
V[1,2] > V[0,2] → item 1 is in the knapsack
→ capacity from 2 to 0

**Matrix Chain Multiplication**

- for finding least amount of scalar multiplication (only cost not how to obtain it)
- M[i,j] = minimum multiplication to compute $A_i A_{i+1} A_{i+2} \cdots A_j$

Matrix has dimension $p_{i-1} \times p_i$ * i as row / j as column *
Thus compute $(A_i \cdots A_k)(A_{k+1} \cdots A_j)$ take: $P_{i-1} \times P_k \times P_j$

$$M[i,j] = \begin{cases} 0, & i = j \\ \min_{i \le k < j} \{ M[i,k] + M[k+1,j] + P_{i-1} \times P_k \times P_j \}, & i < j \end{cases}$$

* i > j is undefined.

ex

| | |
|---|---|
| $A_1$ | 13 × 5 |
| $A_2$ | 5 × 89 |
| $A_3$ | 89 × 3 |
| $A_4$ | 3 × 34 |

| | 13 | 5 | 89 | 3 | 34 |
|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 |

ans 2856 for $A_1A_2A_3A_4$

| i \ j | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 5785 | 1530 | 2856 |
| 2 | X | 0 | 1335 | 1845 |
| 3 | X | X | 0 | 9078 |
| 4 | X | X | X | 0 |

M[1,2] = 0 + 0 + 13×5×89
M[2,3] = 0 + 0 + 5×89×3
M[3,4] = 0 + 0 + 89×3×34
M[1,3] = min{0 + 1335 + 13×5×3, 5785 + 0 + 13×89×3}
M[2,4] = min{0 + 9078 + 5×89×34, 1335 + 0 + 5×3×34}
M[1,4] = min{0 + 1845 + 13×5×34, 5785 + 9078 + 13×89×34, 1530 + 0 + 13×3×34}

DES227
Algorithm Design : Lecture 8-9 (P&NP - Copy w/ Algorithm)

■ P&NP Problems (Lecture 8)

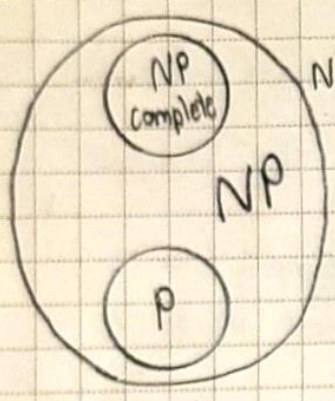
*some take ref. from MIT OCW 6.0465 (Spring 2015/Lecture 16)

→ problems that can be solved in polynomial time are called "tractable"

→ problems that can't be solved in polynomial time are called "intractable"

* polynomial good / exponential bad *

• Definition : Class P ⇒ { problems solvable in polynomial time }




→ some decision problem cannot be solved at all by any algorithm is called undecidable
↳ most famous : halting problem
→ given computer program & input determine program will halt or continue working

Class NP ⇒ { decision problem solvable in polynomial nondeterministic time }
→ equivalent to polynomial time verifier of polynomial-size certificate for YES answer.
→ most decision problems are NP ex hamiltonian circuit, traveling salesman, graph coloring

• Nondeterministic Algorithm ⇒ 2 stages procedure of decision problem

→ Guessing stage : string S can be generated that can be thought of as a candidate solution to the given instance I

→ Verification stage : take I and S as input and present YES if S is solution to instance I

→ said to be nondeterministic polynomial if time efficiency of its verification stage is polynomial.

• Problem X is ⇒ NP-complete if X ∈ NP and X is NP-hard
⇒ NP-hard if every problem Y ∈ NP reduces to X
- if P ≠ NP then X ∉ P

• Reduction from problem A to problem B = polynomial-time algorithm converting A input equivalent (same YES/NO) B inputs.

- if B ∈ P then A ∈ P / - if B ∈ NP then A ∈ NP
- if B ∈ NP-hard then A ∈ NP-hard

- Cyclic thing making CSM reduced to SM problem

• How to prove X is NP-complete
1. belongs to class NP
2. every problem in NP is polynomially reducible to D

* NP-complete ∈ NP, difficult as any other, can be reduce to polynomial time

## Coping w/ limitation of algorithm power (Lecture 9)

- 2 Design techniques
  - ↳ Backtracking
  - ↳ Branch & bound

→ can be though as improvement over exhaustive search, construct candidate sol$^n$ one at a time, evaluate the partially constructed solution

→ worst-case, we still face the same curse of exponential explosion encountered in exhaustive search

→ both techniques based on the construction of "state-space tree"

→ Branch & Bound only to optimization problems
b/c based on computing a bound on possible values of problem's objective function

→ Backtracking is applied to nonoptimization problem

**→ Backtracking**

→ if a partially constructed solution can develop further w/o violating the problem constraints, it is done by taking the first remaining legitimate option for the next component.

→ if there is no legitimate option, no alternatives for remaining, algorithm back track to replace the last component of partially constructed solution with its next option

→ root ⇒ where solution begin

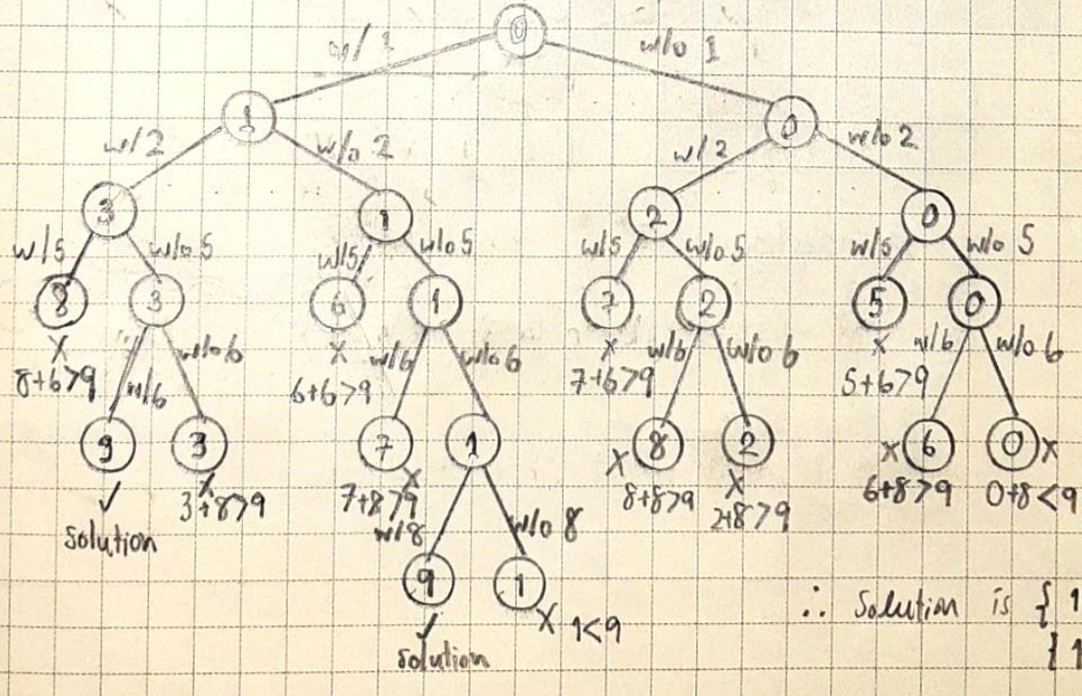
→ node is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise called nonpromising.

**→ n Queens problem** → place n queens on n×n chessboard so that queens not "attacking" each others
→ n=1, trivial sol$^n$ → n=2 & n=3 no solution → n=5 (turn back of this notebook)

**→ Hamiltonian circuit problem** → find Hamiltonian circuit in graph.

**→ Subset-sum problem** → find a subset of a given set S of n positive integers whose sum is equal to given positive integer d.
→ break promising when sum is too large or sum is too small

ex S={1, 2, 5, 6, 8} and d = 9




∴ Solution is {1, 2, 6} and {1, 8}

→ Branch & Bound Technique

→ feasible solution with the best value of objective function is an optimal sol
↳ point in the problem's search space that satisfy all problem's constraint

→ Compared to backtracking, branch-and-bound requires two additional items

1) way to provide a bound on the best value of the objective function on any solution that can be obtained by adding further components to partially constructed solution represented by the node.

2) The value of the best solution seen so far.

→ assignment problem

→ assign n people to n jobs so that the cost is minimum as possible,

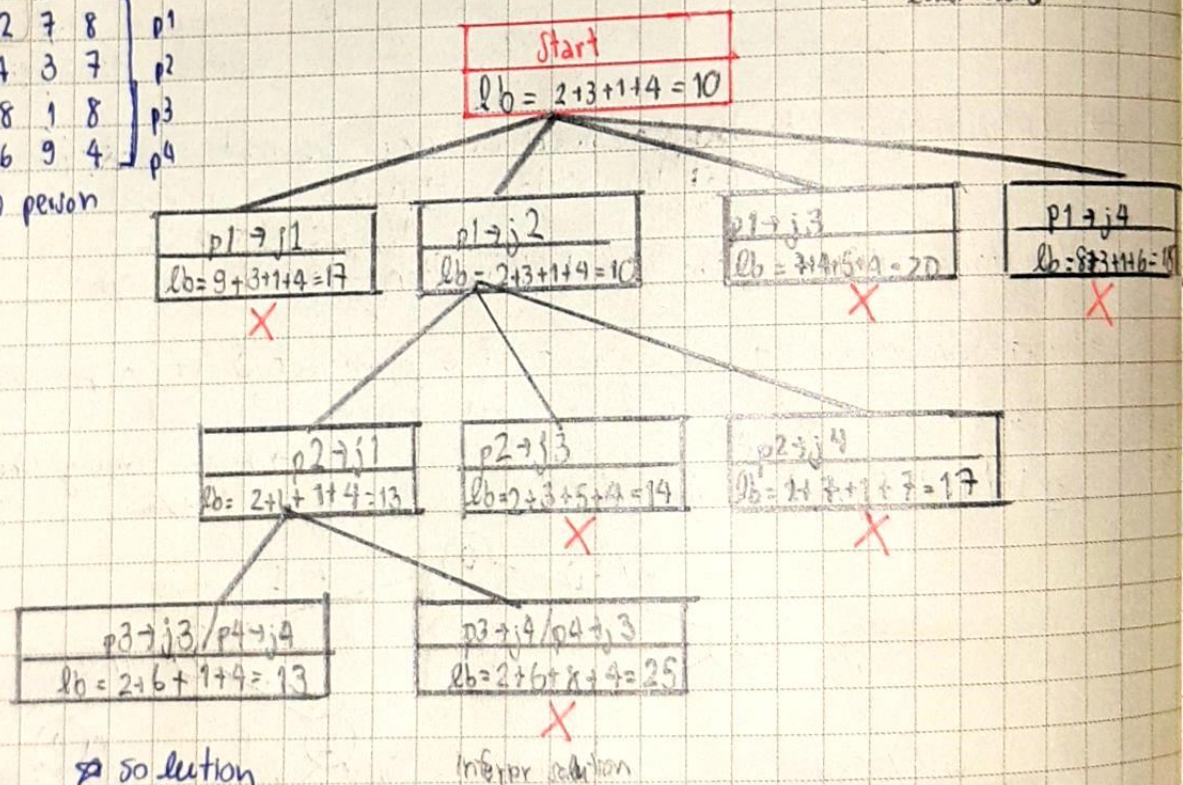
→ starting lower bound → minimum of all job ×

→ used best-first branch-and-bound
→ rather than generate every single child of last promising node as in backtracking, generate all children of the most promising node among live nodes
leaves in current tree, comparing lower bound of live nodes

ex $C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$ (j1 j2 j3 j4 / p1 p2 p3 p4)

j ⇒ job , p ⇒ person

lb = lower bound




✅ solution

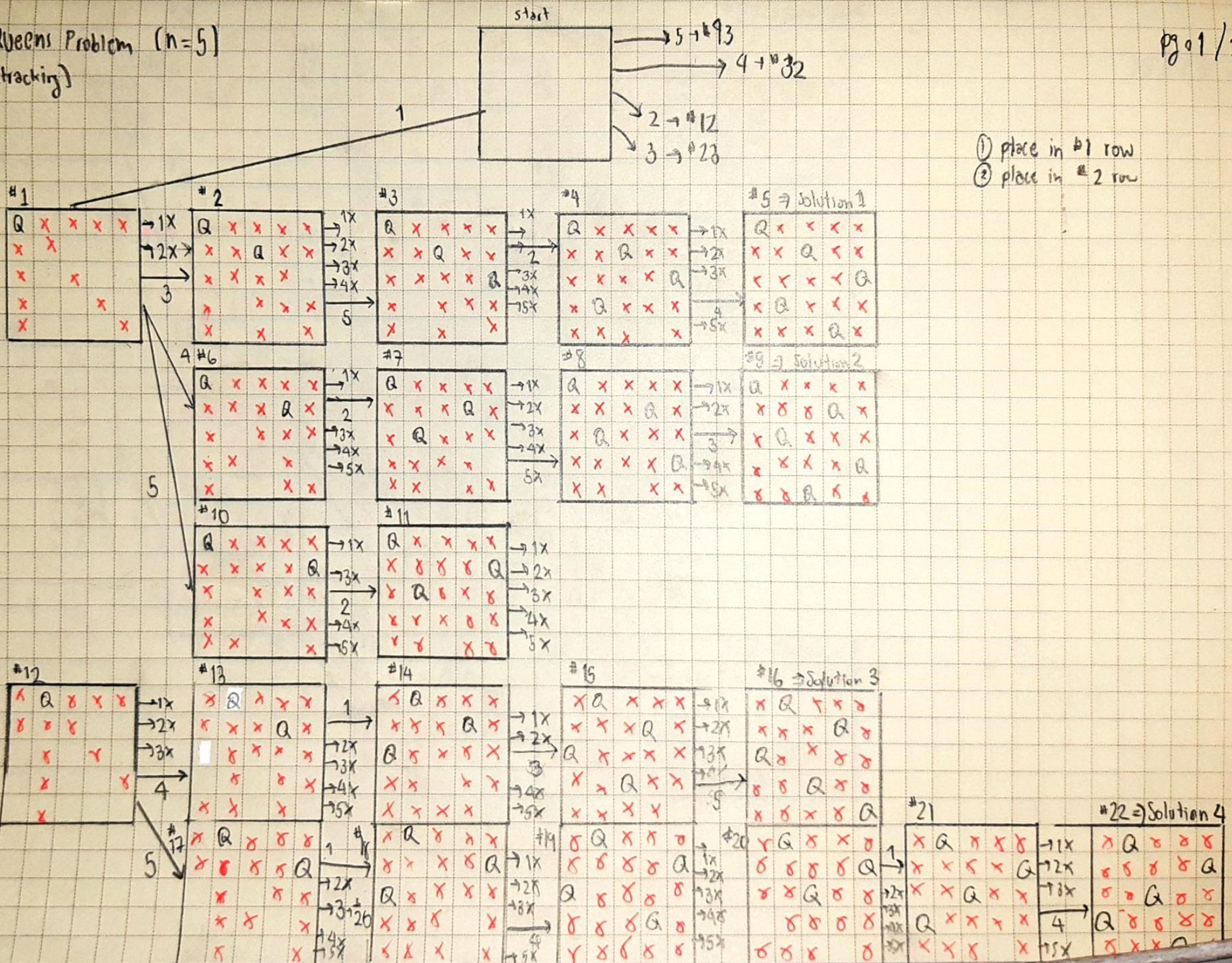
inferior solution

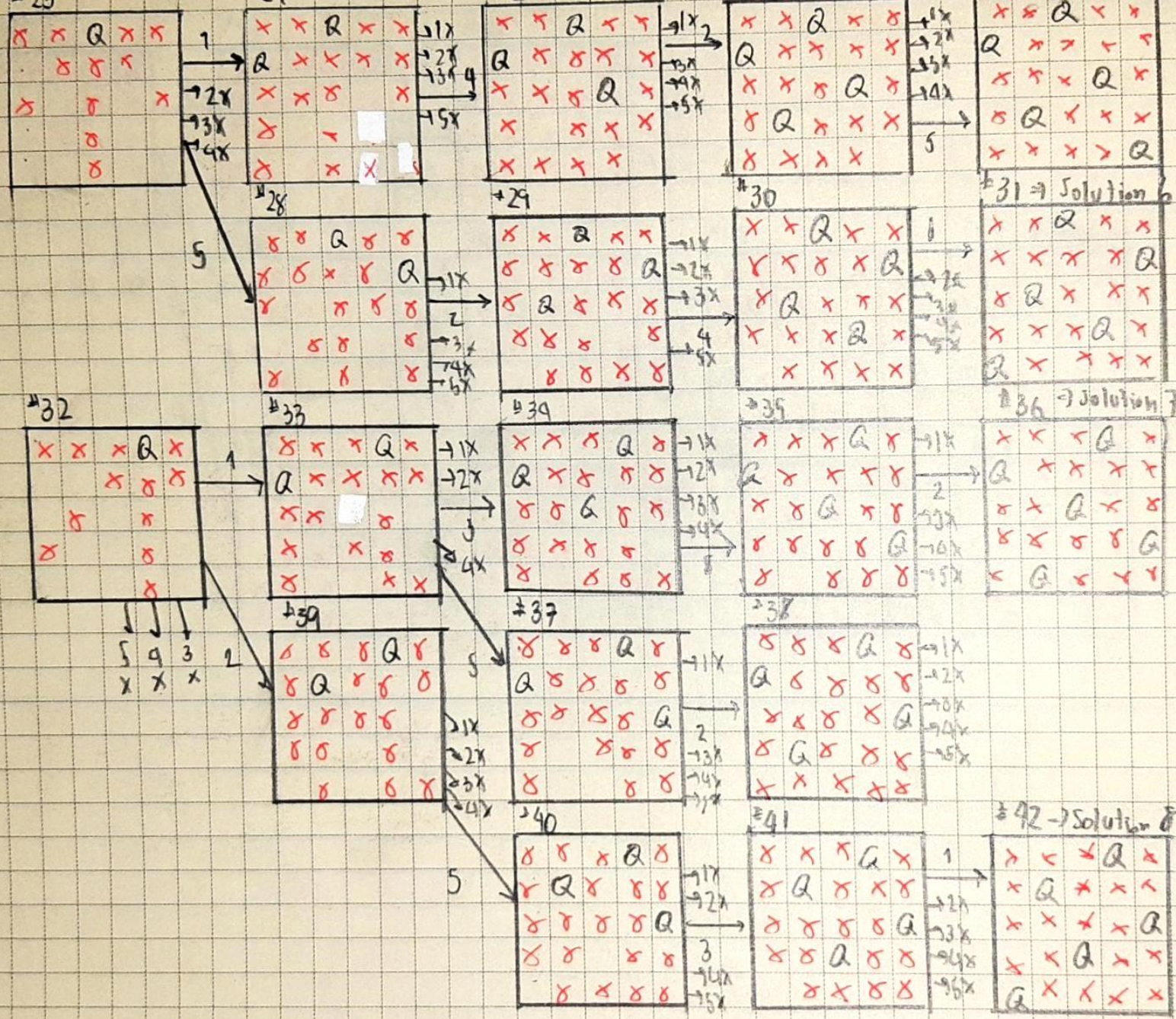
ans {p1 → j2, p2 → j1, p3 → j3, p4 → j4}

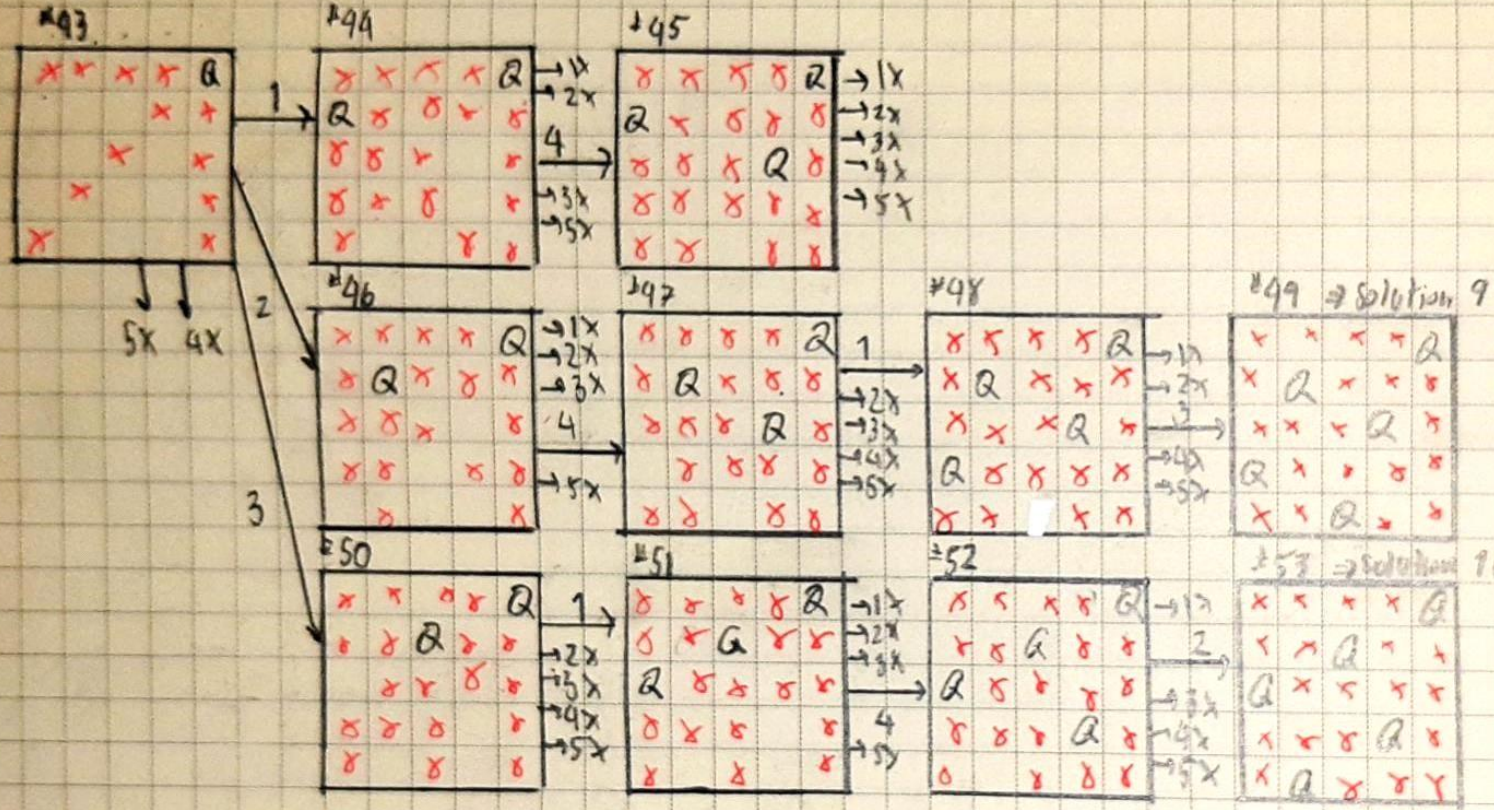
// there's polynomial time algorithm of this problem call Hungarian method.
// branch bound is not for practical use

# N-Queens Problem (n=5)

(Backtracking)




#28
#29
#30
#31 → Solution 6
#32
#33
#34
#35
#36 → Solution 7
#37
#38
#39
#40
#41
#42 → Solution 8


#93
#94
#95
#96
#97
#98
#99 ⇒ Solution 9
#50
#51
#52
#53 ⇒ Solution 10
1
2
3
4
5X
4X
1X
2X
3X
4X
5X