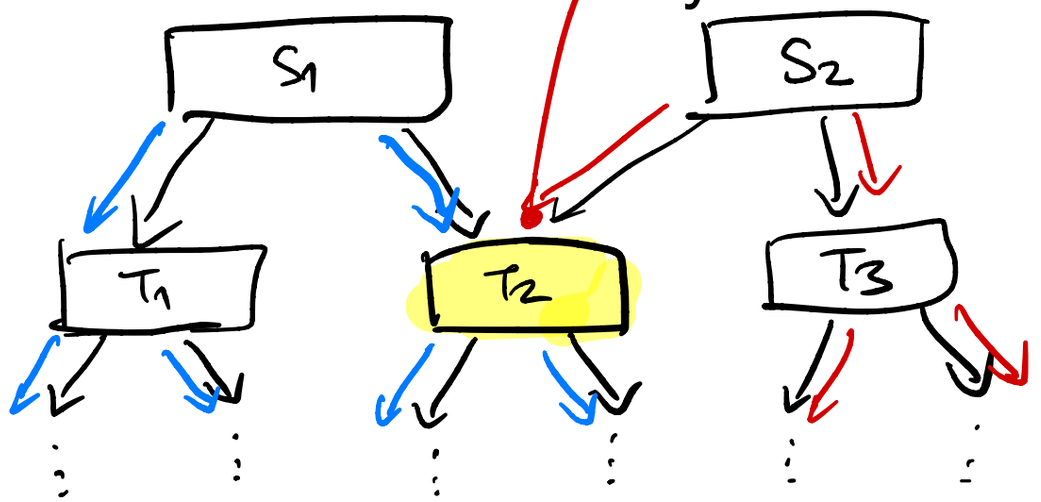
Dynamic programming

Ibrahim Numanagić

Introduction

- Dynamic programming is a fancy name for complete search (brute force) that "remembers" the already seen states



- Key ideas:
  - Find how to encode substates of the problem
  - Make sure that substates / subtrees are independently solvable
  - Make sure to remember already seen substates (e.g. hashmap)

Introduction

- When to use?
  - Anything that requires counting (count the number of ways to do X or Y)
  - Anything that requires optimization (find the largest / smallest item, best / worst score etc.)
  - Anything that requires optimality (find the optimal cut) and that can be done by enumerating all possibilities

Two DP modes

- Top-down DP:
  - Just do standard recursion, remember what you have seen
  - Hint: sub states are usually the recursion parameters that keep changing (if your recursion is correct)
- Bottom-up DP:
  - Tabular method: fill the table from the smallest substate to the largest one
  - Reversed top-down approach
- Neither is "better"
  - Top-down is easier to reason about and might use less memory
  - Bottom-up might be faster as you do not waste time on recursion calls

1D optimization: BABTWR

- BABTWR
  - n different types of blocks are available. Each one of them may be duplicated as many times as you like. Each type has a height y, a width x and a depth z. The blocks are to be stacked one upon each other so that the resulting tower is as high as possible. Of course the blocks can be rotated as desired before stacking. However for reasons of stability a block can only be stacked upon another if both of its baselines are shorter.
- Input
  - The number of types of blocks n is located in the first line of each test case. On the subsequent n lines the height yi, the width xi and the depth zi of each type of blocks are given. There are never more than 30 different types available. There are many test cases, which come one by one. Input terminates with n = 0. Edited: You can assume that max(xi, yi, zi) <= 2500.
- Output
  - For each test case your program should output one line with the height of the highest possible tower.

Sample input:

```
5
31 41 59
26 53 58
97 93 23
84 62 64
33 83 27
1
1 1 1
0
```

Sample output:

```
342
1
```

1D optimization: BABTWR

Notes:

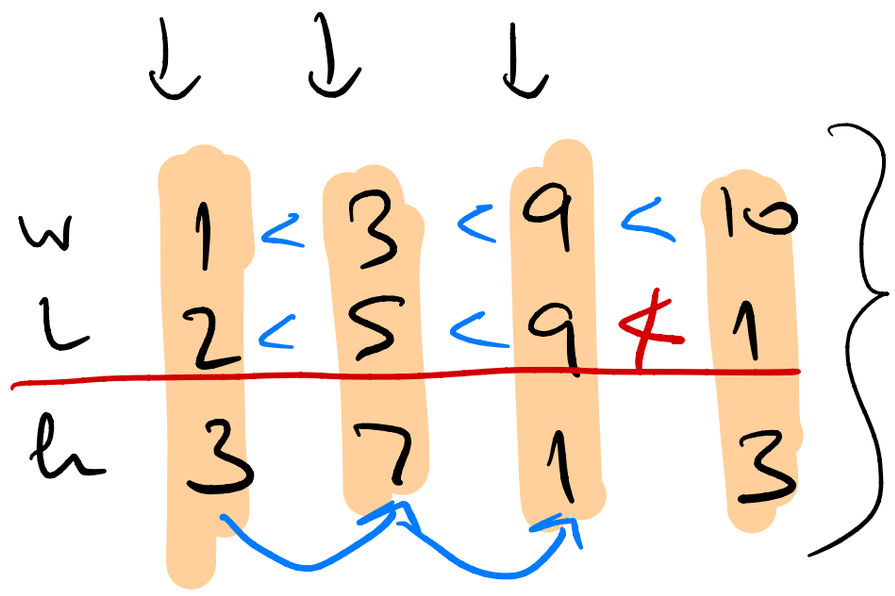
① each block can be used max. 3 times (3 rotations) due to stacking constraint

② form a list L of ALL rotations (3 of them) of each cube

③ Sort L

④ do a variant of LIS on L ($O(n^2)$ is enough to get AC)

(only w and l matter for LIS; h is used for the sum



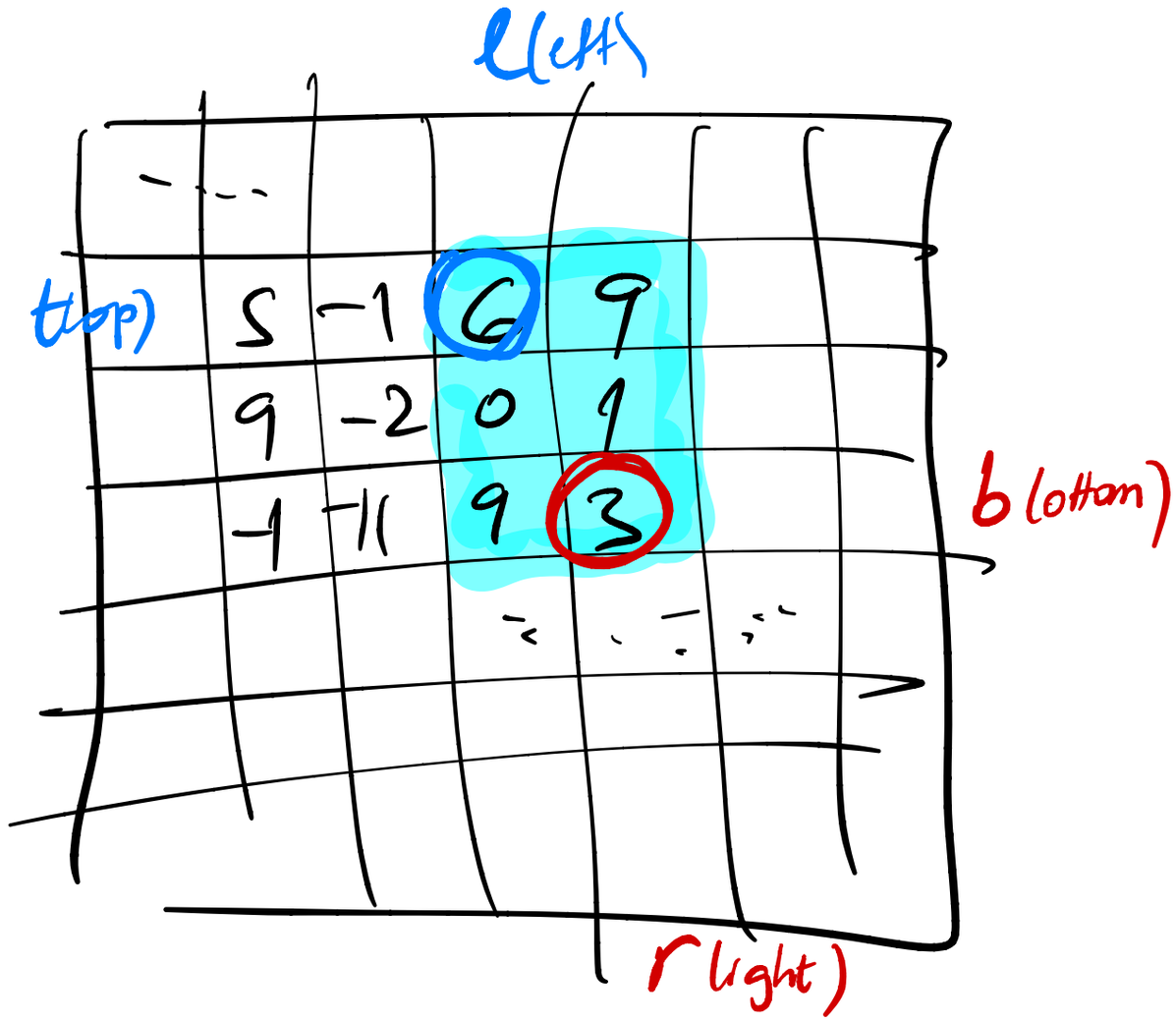
List L

(do LIS on w & L, keep sum of h's as the "length")

2D counting: cumulative sums

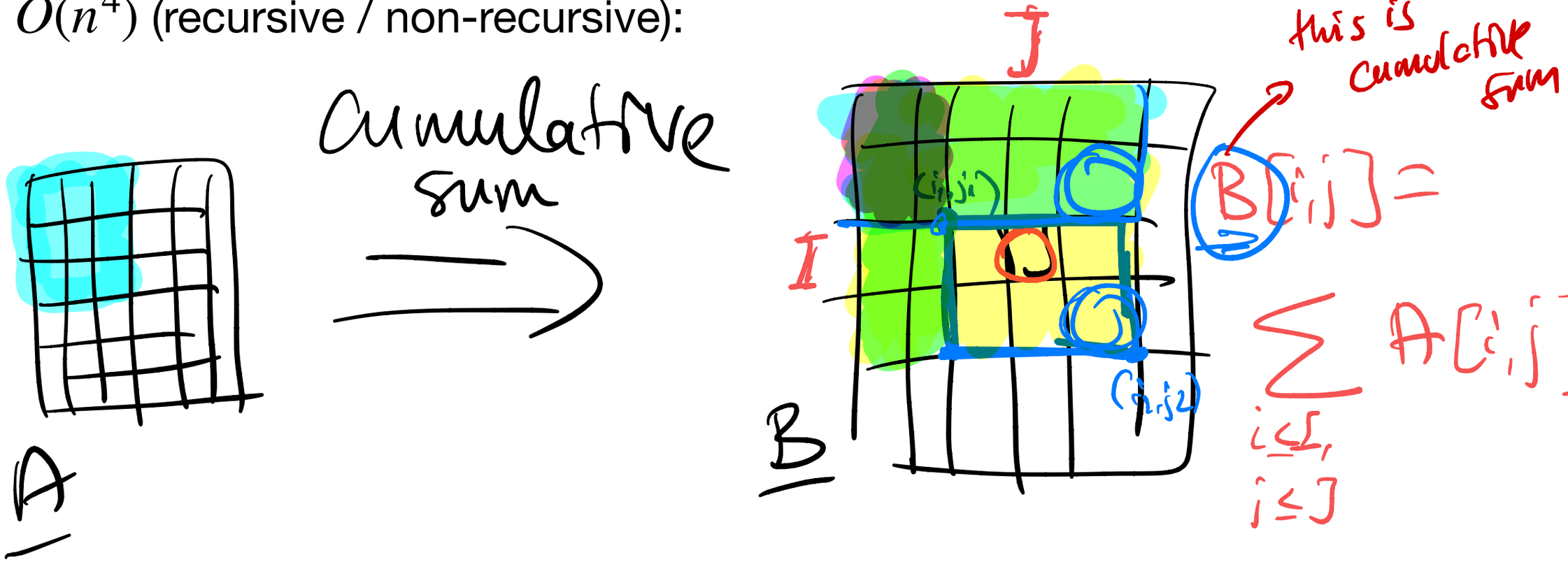
- UVa 108: https://onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show_problem&problem=44
- $O(n^6)$:

for each (t,l)
  for each (b,r)
    sum matrix [(t,l) → (b,r)]



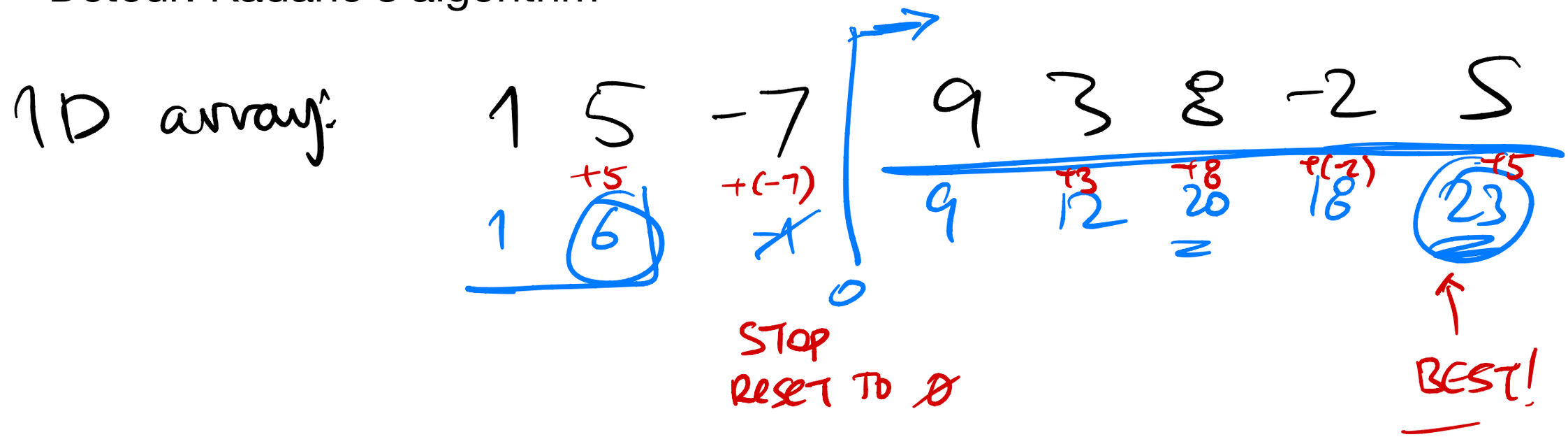
2D counting: cumulative sums

- UVa 108: https://onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show_problem&problem=44
- $O(n^4)$ (recursive / non-recursive):



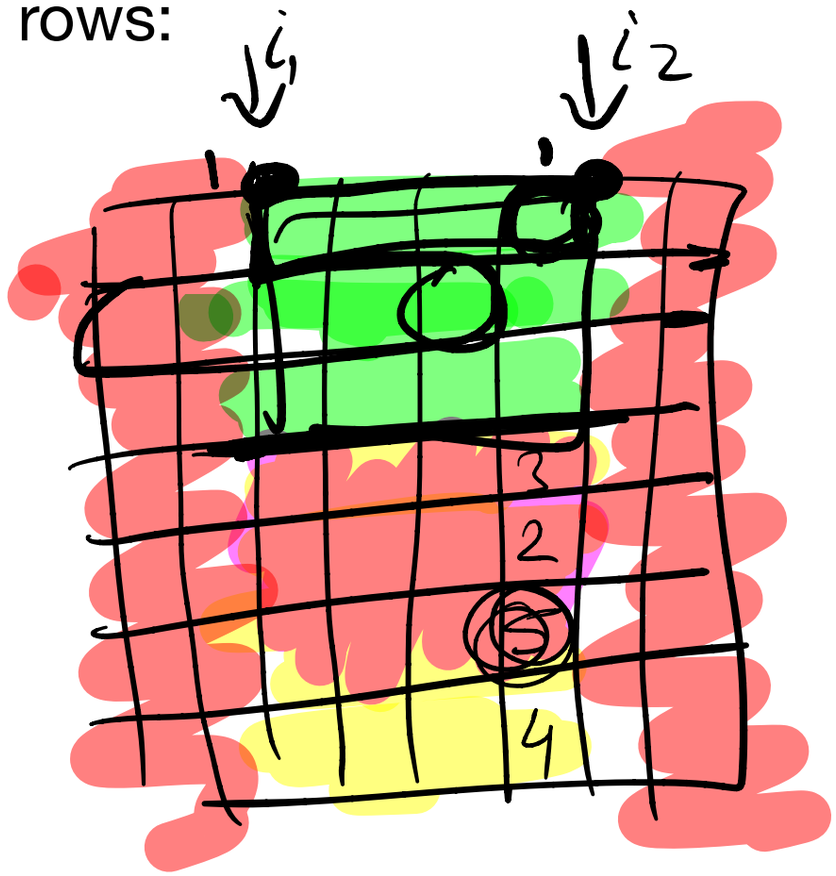
2D counting: cumulative sums

- UVa 108: https://onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show_problem&problem=44
- Detour: Kadane's algorithm



2D counting: cumulative sums

- UVa 108: https://onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show_problem&problem=44
- $O(n^3)$ Kadane on rows:



Kadane on rows!

Cumulative sum of ROWS!

$$B[i,j] = \sum_{j' \le j} A[i,j']$$

ALG:

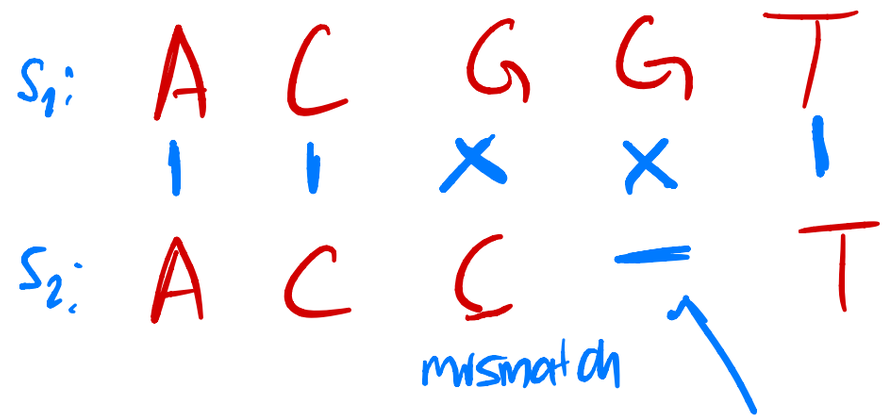
```
for column i1
  for column i2
    for row r
      if sum + [B[r,i2] - B[r,i1]] ≥ 0:
        sum += B[r,i2] - B[r,i1]
        check if this is the largest sum so far
      else:
        sum = 0
```

2D counting: cumulative sums

- Variations:
  - Find a submatrix with smallest mean
  - Find a submatrix with all ones if input is 0/1 matrix
  - Find a largest submatrix by area containing all ones if input is 0/1 matrix
  - Find a largest square submatrix containing all ones if input is 0/1 matrix
  - Find the number of submatrices containing all zeros (or ones) if input is 0/1 matrix (Google Interview)

2D and 3D optimization: edit distance

- Given two strings, find minimum number of changes needed to convert the first into the second
- https://onlinejudge.org/external/131/13146.pdf
- Subproblems: substrings



Recurrence:

$$ed[i,j] = \min \begin{cases} ed[i-1,j-1] + \begin{cases} 1 & \text{if } s_1[i] \neq s_2[j] \\ 0 & \text{otherwise} \end{cases} \\ ed[i-1,j] + 1 \\ ed[i,j-1] + 1 \end{cases}$$

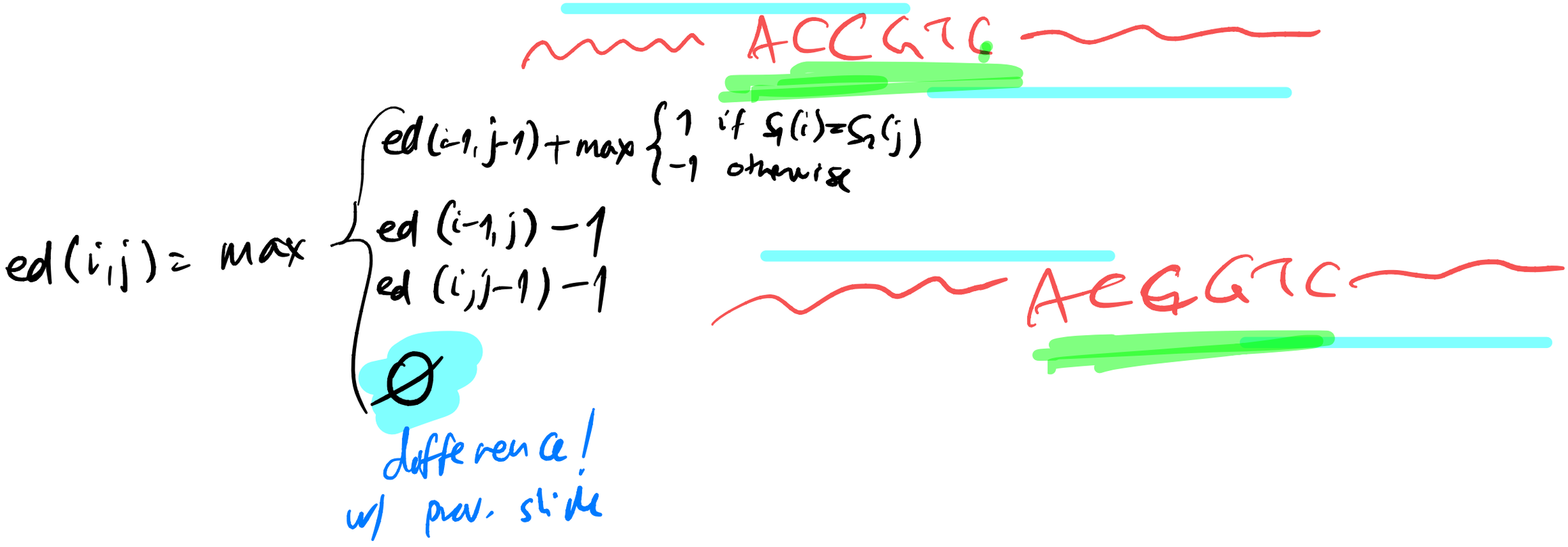
$\Rightarrow ed(s_1, s_2) = \underline{\underline{\leq}}$

$ed[i,j]$ is edit distance of $s_1[1 \ldots i]$ and $s_2[1 \ldots j]$

(SUBPROBLEMS: prefixes of s_1 and s_2)

2D and 3D optimization: edit distance

- Given two strings, find minimum number of changes needed to convert the first into the second
- https://onlinejudge.org/external/131/13146.pdf
- Change: find the closest substrings!

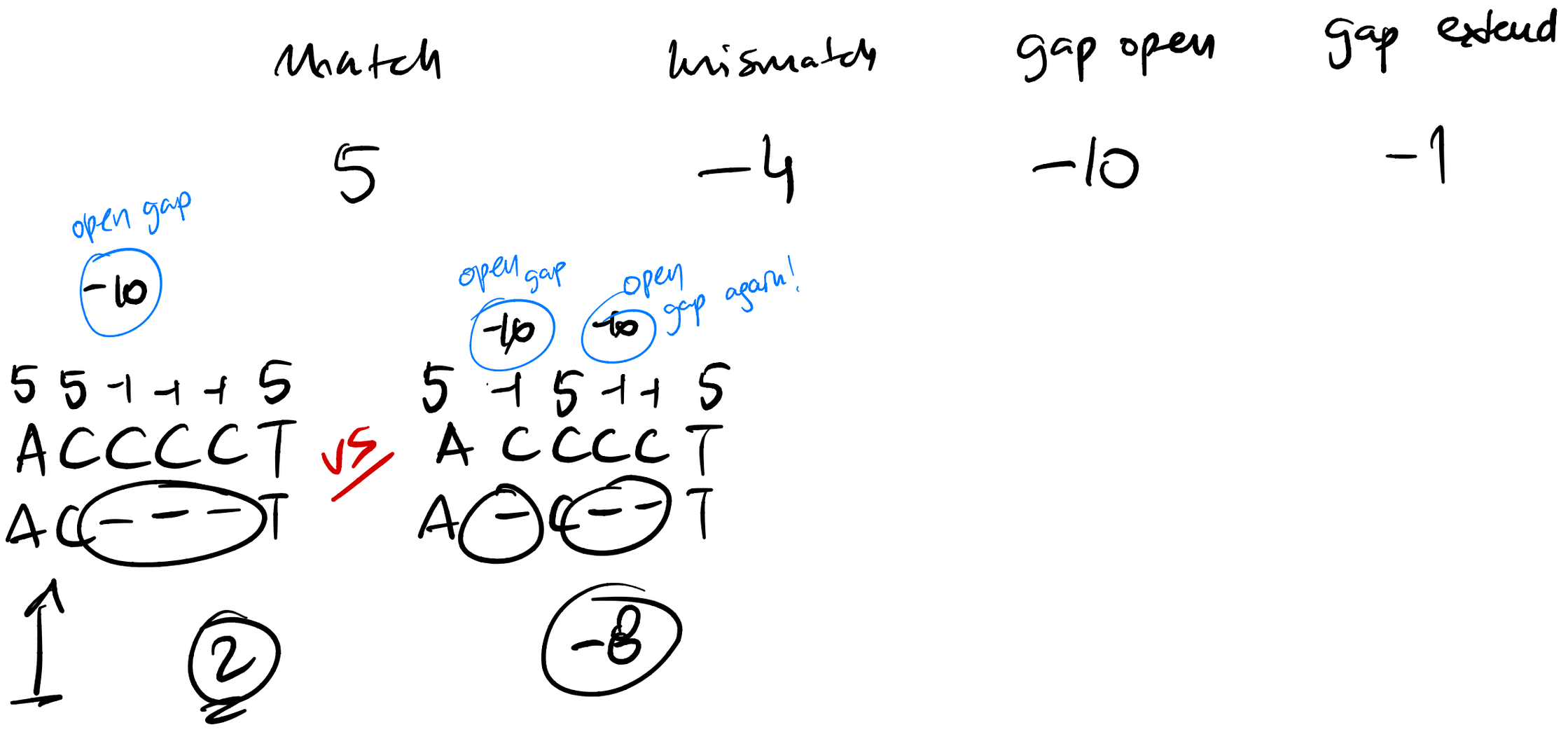


2D and 3D optimization: edit distance

- Reconstruction:

2D and 3D optimization: edit distance

- Given two strings, find minimum number of changes needed to convert the first into the second
- Change: gaps can be affine (e.g. we have gap open and gap extension cost)!



2D and 3D optimization: edit distance

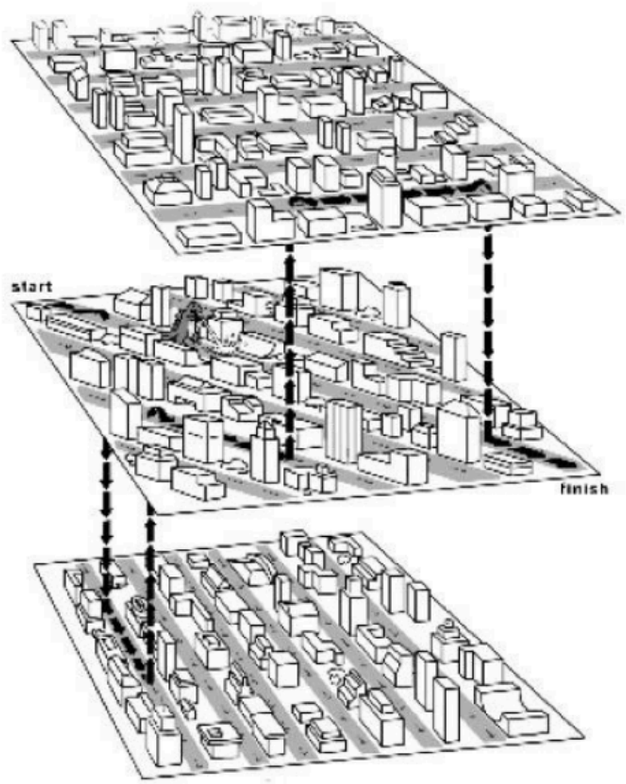
See for more examples:
https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/gaps.pdf

- Given two strings, find minimum number of changes needed to convert the first into the second
- Change: gaps can be affine (e.g. we have gap open and gap extension cost)!

$$dp(i,j) \overset{\max}{=} \begin{cases} dp[i-1,j-1] + \text{match or mismatch} & \text{dep. on } s_1[i] \neq s_1[j] \\ ig[i,j] \\ dg[i,j] \end{cases}$$

$$ig(i,j) \overset{\max}{=} \begin{cases} ig(i-1,j) + \text{gap\_extend} \\ dp(i-1,j) + \text{gap\_open} + \text{gap\_extend} \end{cases}$$

$$dg(i,j) = \max \begin{cases} dg(i,j-1) + \text{gap\_extend} \\ dp(i,j-1) + \text{gap\_open} + \text{gap\_extend} \end{cases}$$



2D and 3D optimization: edit distance

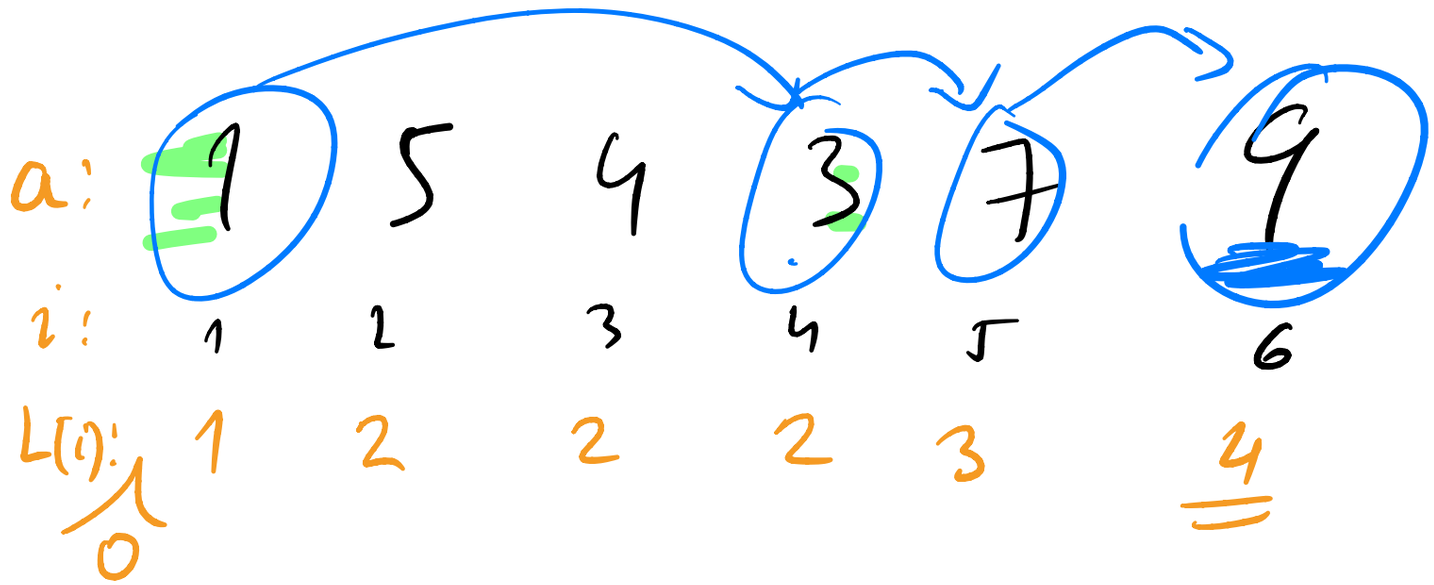
- Similar problems:
  - Longest common subsequence (mismatch is infinity)
  - https://onlinejudge.org/external/100/p10029.pdf
  - Most of the problems with typo detection, genome/DNA strings, string similarity etc.
  - The whole field of bioinformatics

1D optimization: LIS

- Find longest increasing subsequence
- https://onlinejudge.org/external/127/p12747.pdf
- $O(n^2)$:

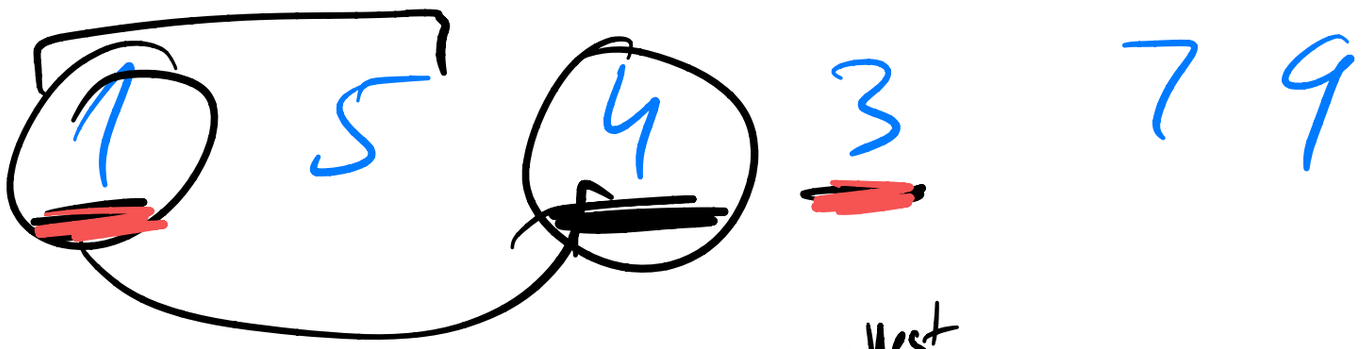
$L(0) = 0$

$L(i) = \max_j \{ L[j]+1 \text{ if } a[i] > a[j] \}$



1D optimization: LIS

- Find longest increasing subsequence
- https://onlinejudge.org/external/127/p12747.pdf
- $O(n \log n)$:



L

1	1 < 4
2	3
3	7
4	9
5	∞
6	∞

① L[i] keeps the smallest last element of any LIS of length i found thus far

② Clearly L is sorted

⇒ Can use binary search to get $O(n \log n)$

Pseudo-polynomial memoization

- Coin change (https://onlinejudge.org/external/6/674.pdf)
- Similar to counting problems, but the size of the table is determined by the value

- pseudo-polynomial!
complexity is $O(nV)$
where V is max. value
(V is NOT polynomial on n!)

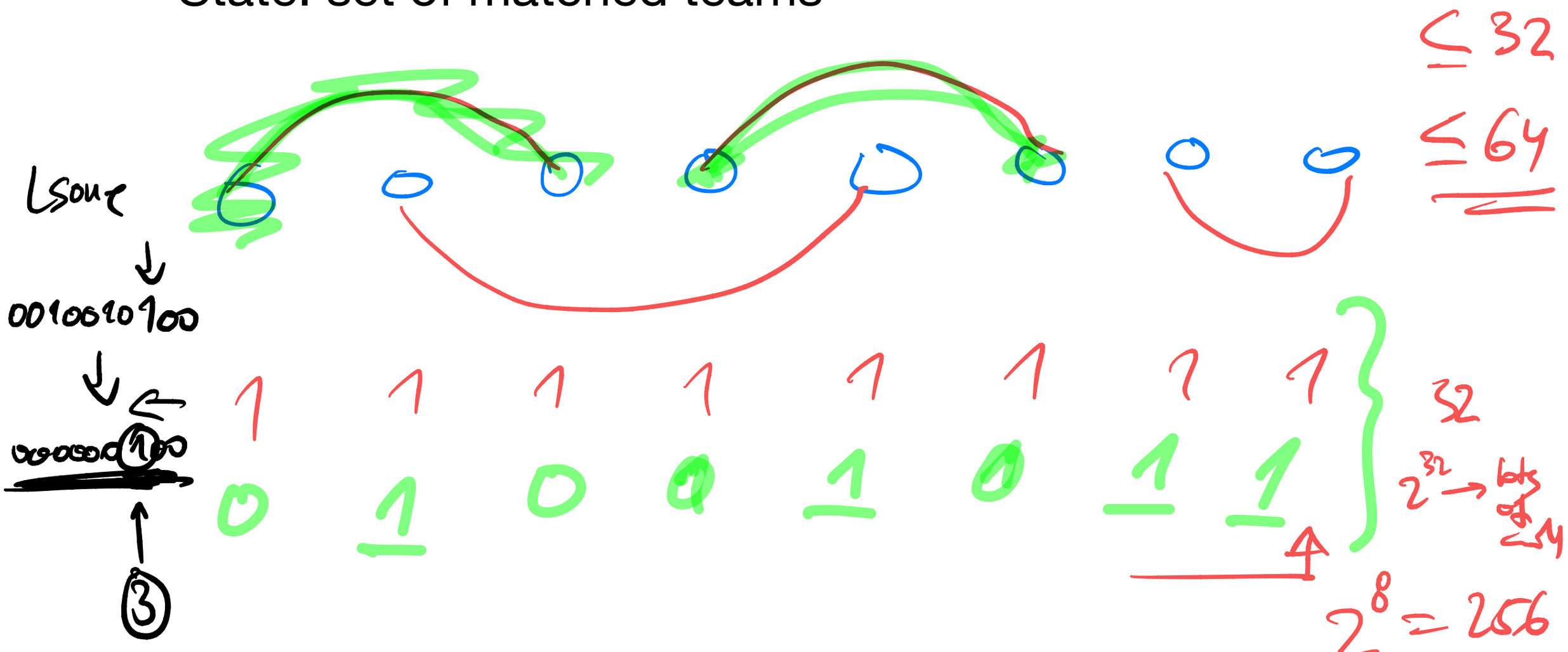
[see the code for details]

Pseudo-polynomial memoization

- Similar problems:
  - Counting number of ways to get the change (extra state: type)
  - All sorts of knapsack

Bitmask states

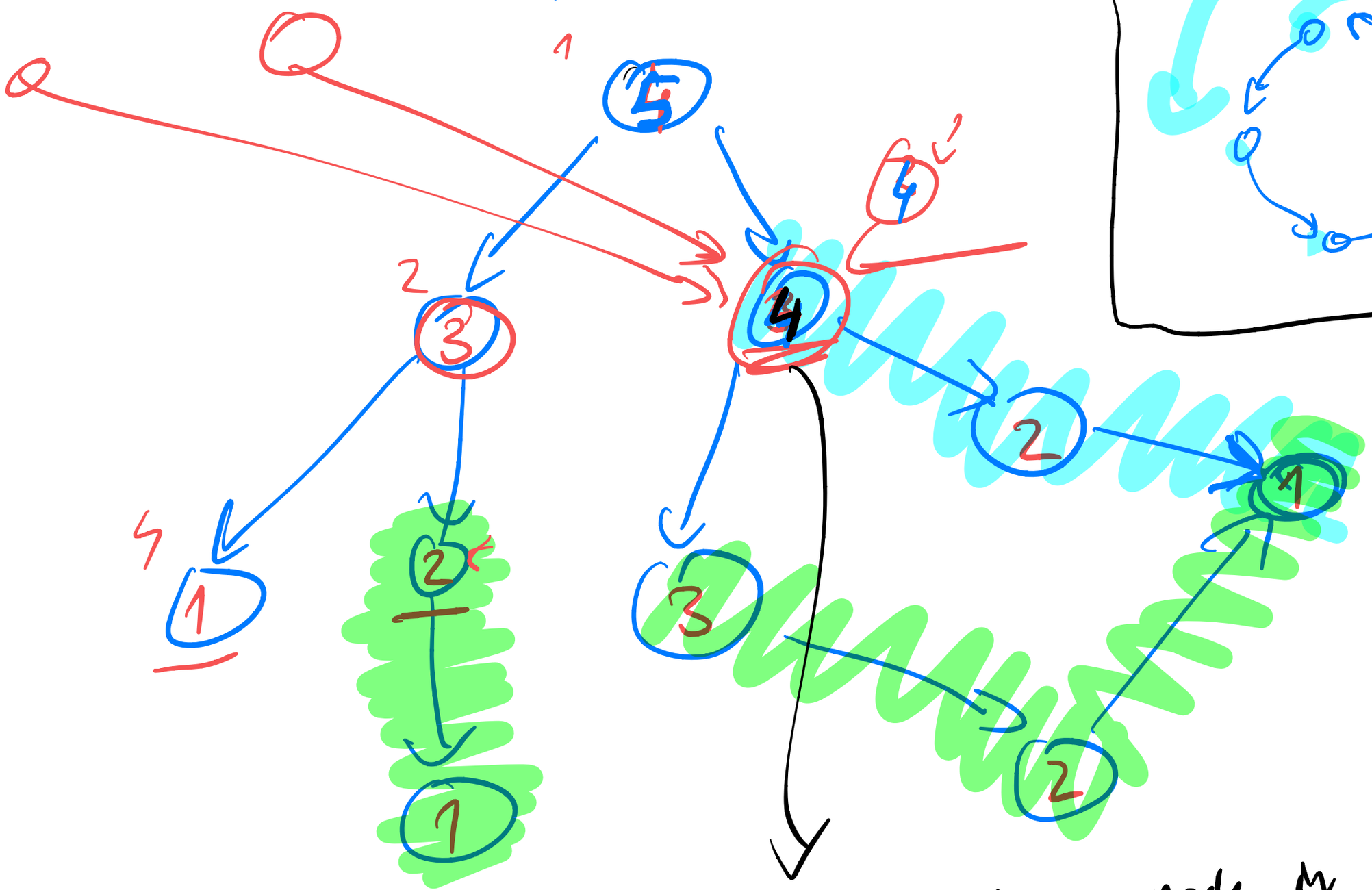
- Problem https://onlinejudge.org/external/109/10911.pdf
- State: set of matched teams

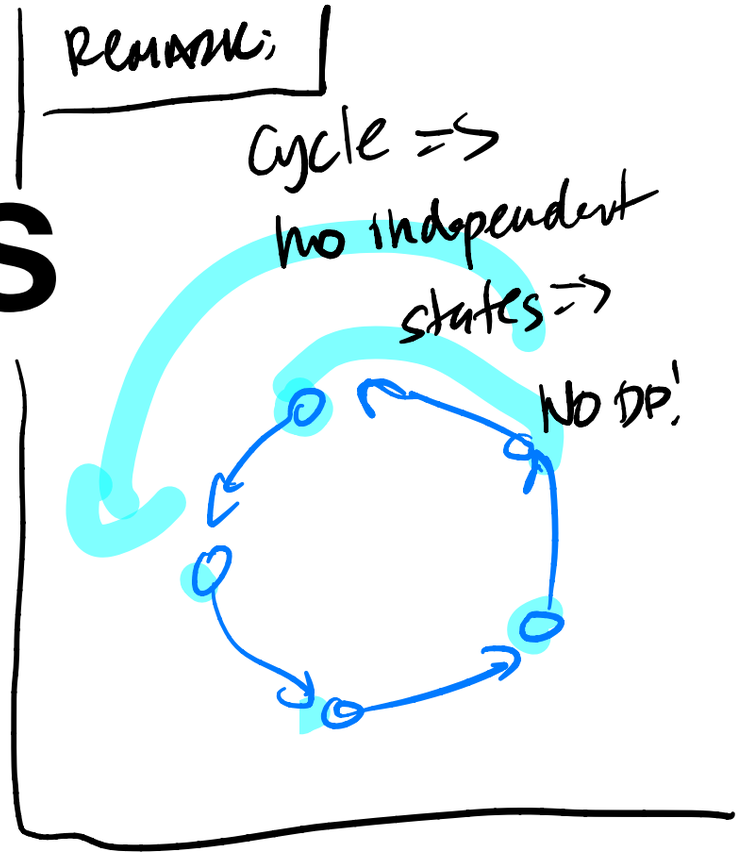


Bitmask states

- Problem https://onlinejudge.org/external/109/10911.pdf
- State: set of matched teams

DAG states





DP state is a node in a dir. acyclic graph!

1. Parameter: Index i in an array, e.g. $[x_0,\ x_1,\ \ldots,\ x_i,\ \ldots]$
   Transition: Extend subarray $[0..i]$ (or $[i..n\text{-}1]$), process i, take item i or not, etc
   Example: 1D Max Sum, LIS, part of 0-1 Knapsack, TSP, etc (see Section 3.5.2)

2. Parameter: Indices (i, j) in two arrays, e.g. $[x_0,\ x_1,\ \ldots,\ x_i] + [y_0,\ y_1,\ \ldots,\ y_j]$
   Transition: Extend i, j, or both, etc
   Example: String Alignment/Edit Distance, LCS, etc (see Section 6.5)

3. Parameter: Subarray (i, j) of an array $[\ldots,\ x_i,\ x_{i+1},\ \ldots,\ x_j,\ \ldots]$
   Transition: Split (i, j) into $(i, k) + (k + 1, j)$ or into $(i, i + k) + (i + k + 1, j)$, etc
   Example: Matrix Chain Multiplication (see Section 9.20), etc

4. Parameter: A vertex (position) in a (usually implicit) DAG
   Transition: Process the neighbors of this vertex, etc
   Example: Shortest/Longest/Counting Paths in/on DAG, etc (Section 4.7.1)

5. Parameter: Knapsack-Style Parameter
   Transition: Decrease (or increase) current value until zero (or until threshold), etc
   Example: 0-1 Knapsack, Subset Sum, Coin Change variants, etc (see Section 3.5.2)
   Note: This parameter is not DP friendly if its range is very high.
   See tips in Section 8.3.3 if the value of this parameter can go negative.

6. Parameter: Small set (usually using bitmask technique)
   Transition: Flag one (or more) item(s) in the set to on (or off), etc
   Example: DP-TSP (see Section 3.5.2), DP with bitmask (see Section 8.3.1), etc

Do all the problems here

- CP1 3.5
- CP2 8.3

} check all of them!