CS2020 – Data Structures and Algorithms Accelerated

Lecture 21 – DP, the True Form

stevenhalim@gmail.com

National University of Singapore

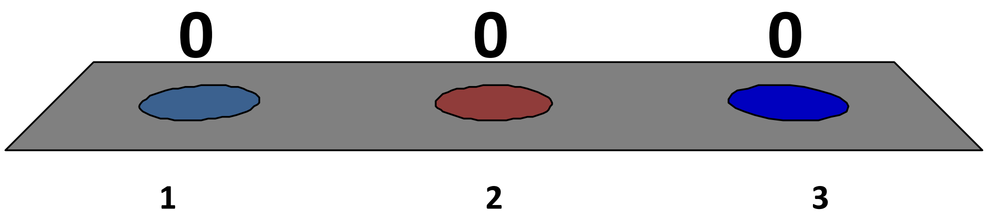
School of Computing

Outline

- What are we going to learn in this lecture?
  - Review (DP for TSP from previous lecture)
  - Presentation of several classical problems solvable with DP technique (not natural to be viewed as a graph problem)
    - 1-D Range Sum (Isn’t this a **math** problem?)
    - 0-1 Knapsack, a pseudo-polynomial algorithm
    - String Alignment (DP on **String**, can we run DP on string?)
    - Outline of TUTOR (PS10)
  - Troughout lecture:
    - Discussion of distinct states (vertices on implicit DAG)
& its space complexity
    - Discussion of overlapping transitions (edges on implicit DAG)
& its time complexity

Can you write recursive function involving vertices of a bipartite graph?

1. Why not?
2. Always cannot, because ______
3. Sometimes it is possible, sometimes it is not possible, because _______

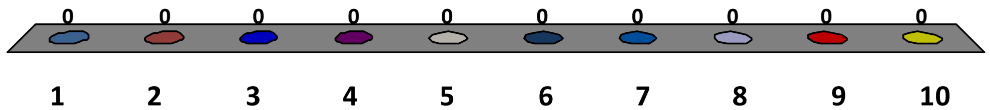


0 of 54

For those who have read & attempt NOI 2011: TOUR, list down the necessary DS/algorithm to solve this problem (each clicker can select up to 5)

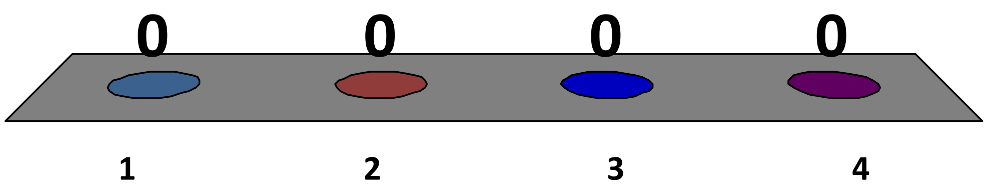
1. Adjacency Matrix
2. Adjacency List
3. No DS, Implicit Graph
4. DFS
5. Backtracking
6. BFS
7. Dijkstra's
8. Bellman Ford's
9. DP
10. Bitmask

0 of 54



And how about the other problem: TUTOR

1. I haven’t download PS10.pdf ☹
2. I have solved TOUR (only)
3. I have solved TUTOR (only)
4. I have solved both TOUR and TUTOR



0 of 54

TSP Review/Clarifications

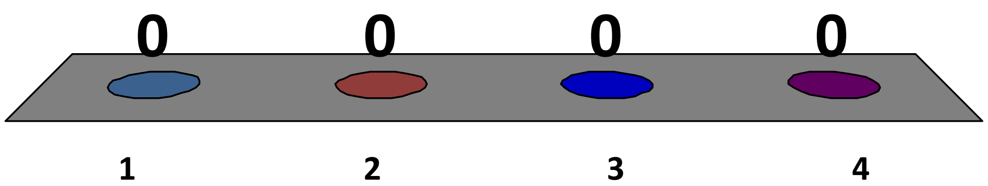
- Let's go back to Lecture20 slides/TSPDemo.java
  - "backtracking v1" is not that slow if I turn off the I/O part
    - But it is still very slow for n > 11
  - We have not discussed "backtracking v2"
    - And the example of "wrong" DP states (memo1)
      - Although it runs "very fast"
    - Plus one bug fix in that backtracking v2 (please take a note)
  - We will do experiment with "DP_TSP" function
    - Turning off the memo check to convert this recursive function back to simple backtracking and see the difference in speed

1-D Max Range Sum

- Given a (1D) **array** of integers of size **N**
  - Example:
    - arr = { 2, -4, 8, 5, -9, 7 }
- Determine the **range sum** from index i to index j? **RS(i, j)**
  - Examples: (note: 0-based indexing)
    - RS(0, 5) = 2 - 4 + 8 + 5 - 9 + 7 = 9
    - RS(2, 3) = 8 + 5 = 13
    - RS(3, 5) = 5 - 9 + 7 = 3
- 1D Max Range Sum:
  Find value of **i** and **j** so that **RS(i, j)** is maximum
  - For this example, i = 2 and j = 3, because RS(2, 3) = 13 is the maximum possible over all ranges

Quick Survey: 1-D Max Range Sum

1. I have not seen this problem before
2. I have seen this problem before
3. I have solved (coded a solution for) this problem before
4. On top of no 3, I also know the 2-D variant



0 of 54

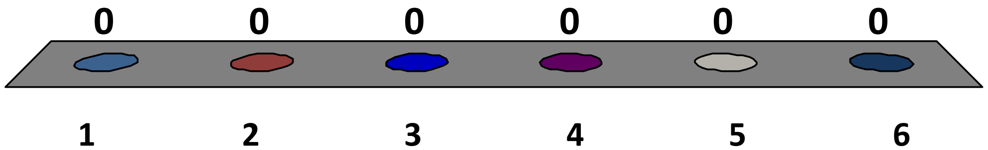
1D Max Range Sum: Naïve Solution

```
maxRangeSum ← arr[0] // pick one val as current best
best_i ← -1
best_j ← -1
for each i ∈ [0..N-2]
  for each j ∈ [i+1..N-1]
    sum ← 0
    for k = i to j
      sum ← sum + arr[k]
    if sum > maxRangeSum
      maxRangeSum ← sum
      best_i ← i
      best_j ← j
```

The Naïve Solution runs in…

1. $O(N)$
2. $O(N \log N)$
3. $O(N^2)$
4. $O(N^2 \log N)$
5. $O(N^3)$
6. $O(N!)$, like TSP

0 of 54



Can we use DP?

- Optimal substructures?
  - Yes, for example we can write this recurrence:

    RS(i, j) = arr[i] + RS(i + 1, j)
- Overlapping subproblems?
  - Yes, for example we can have this situation:
    Both RS(i, b) and RS(a, j) where i < a < b < j
    has to compute RS(a, b)
    - Example: both RS(0, 10) and RS(5, 15) has compute RS(5, 10)

DP Formulation v1

- Distinct States:
  - **Two** parameters: i and j
  - This is the most natural formulation
  - Space complexity: $O(N * N) = O(N^2)$
- Overlapping Transitions:
  - RS(i, j) = arr[i]; **if i == j** (last item)
  - RS(i, j) = arr[i] + RS(i + 1, j); **if i != j**
  - Time complexity: $O(N^2 \times 1) = O(N^2)$
- This is not the most efficient way…

DP Formulation v2

- Distinct States:
  - **One** parameter: i → non trivial
  - Space complexity: O(N)
- Overlapping Transitions:
  - **preprocess(i)** = arr[i]; **if i == 0** (first item)
  - **preprocess(i)** = arr[i] + **preprocess**(i - 1); **if i > 0**
  - Time complexity: O(N x 1) = O(N) for all i $\in$ [0 .. N]
- Then how to compute RS(i, j)?
  - RS(i, j) = preprocess(j); **if i == 0**
  - RS(i, j) = preprocess(j) - preprocess(i - 1); **if i > 0**
  - This is O(1)

1D Max Range Sum: DP Solution v2

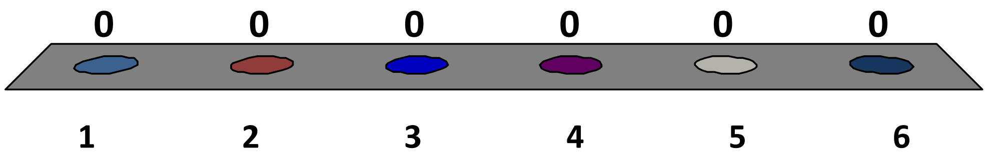
```
maxRangeSum ← arr[0]
best_i ← -1
best_j ← -1
for each i ∈ [0..N-1]
  preprocess(i)
for each i ∈ [0..N-2]
  for each j ∈ [i+1..N-1]
    if RS(i, j) > maxRangeSum
      maxRangeSum ← sum
      best_i ← i
      best_j ← j
// usual implementation: bottom up (topological order)
```

Bottom Up versus Top Down

- Top Down
  - Before entering recursion, check if this state has been computed, if it is, do not recompute
  - Before exiting the recursion, store the computation result in a table
- Bottom Up
  - Prepare a table that will store the values of each sub problems
  - Find a topological order to fill the table so that all smaller sub problems necessary to solve a bigger sub problem have been computed before

The DP Solution v2 runs in…

1. $O(N)$
2. $O(N \log N)$
3. $O(N^2)$
4. $O(N^2 \log N)$
5. $O(N^3)$
6. $O(N!)$, like TSP



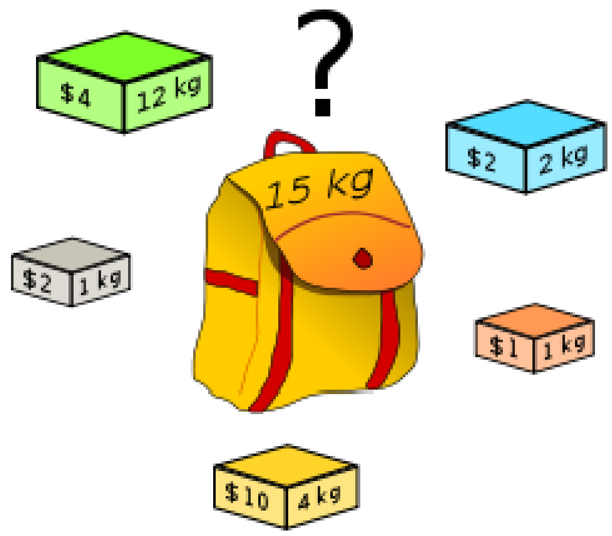
0 of 54

So…

1. That is a pretty impressive improvement, show me the next DP problem
2. Eh wait… I know an even better solution for 1D Max Range Sum

0 of 54

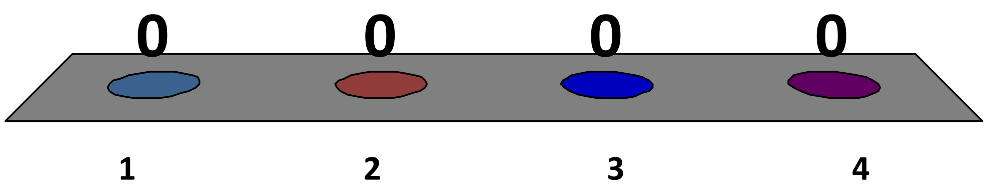
0-1 Knapsack



- Problem Definition:
  - Given a set of items
    - Each item has associated weight and value
    - See the figure on the right
  - Determine which items that we should take (0-1) such that:
    - The total weight is **less than or equal to** the limit of the knapsack
    - The total value is as large as possible

Quick Survey: 0-1 Knapsack

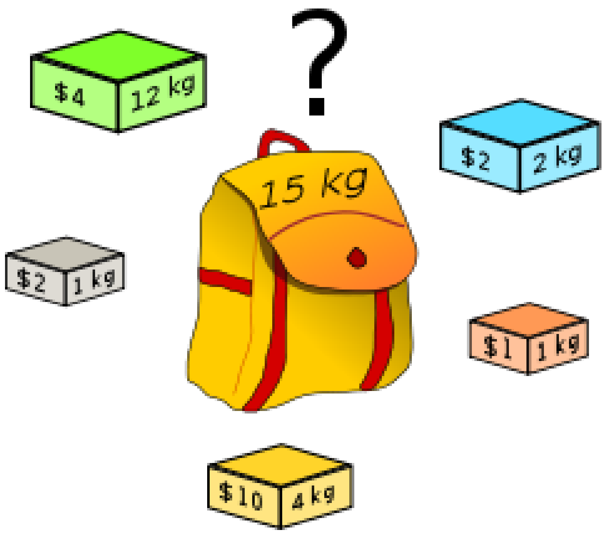
1. I have not seen this problem before
2. I have seen this problem before
3. I have solved (coded a solution for) this problem before
4. On top of no-3, I also know the other variant: the fractional knapsack

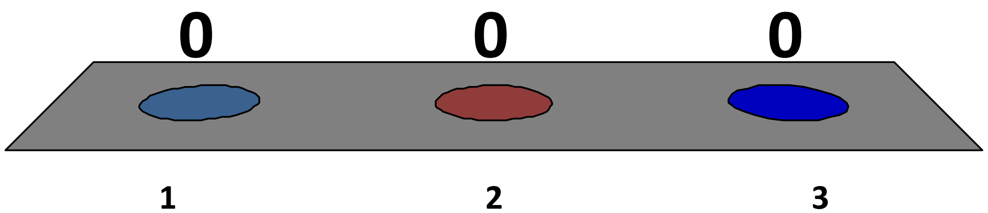


0 of 54

Given this 0-1 Knapsack instance, which items that we should take to maximize the value while satisfying the knapsack constraint?

1. Everything, tot weight = 20kg, tot value = 19$
2. Everything but the green box, tot weight = 8kg, tot value = 15$
3. Green, blue, and grey box, tot weight = 15kg, tot value = 8$





0 of 54

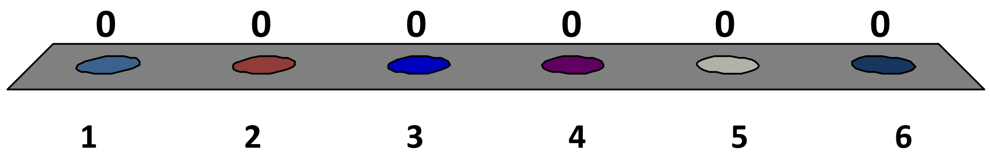
0-1 Knapsack: Naïve Solution

```
maxValue ← 0
bestSet ← 0
for set = 0 to (2^N - 1)
  w ← computeWeight(set)
  if w <= KnapsackSize
    v ← computeValue(set)
    if v > maxValue
      maxValue ← v
      bestSet ← set
```

- Here, we use similar idea as in TSP, using integer to represent set of boolean
- Suppose N = 2
  - set = $0_{10} = 00_2$
  - set = $1_{10} = 01_2$
  - set = $2_{10} = 10_2$
  - set = $3_{10} = 11_2$

The Naïve Solution runs in…

1. $O(N^2)$
2. $O(N^2 \log N)$
3. $O(N^3)$
4. $O(2^N)$
5. $O(N * 2^N)$
6. $O(N!)$, like TSP



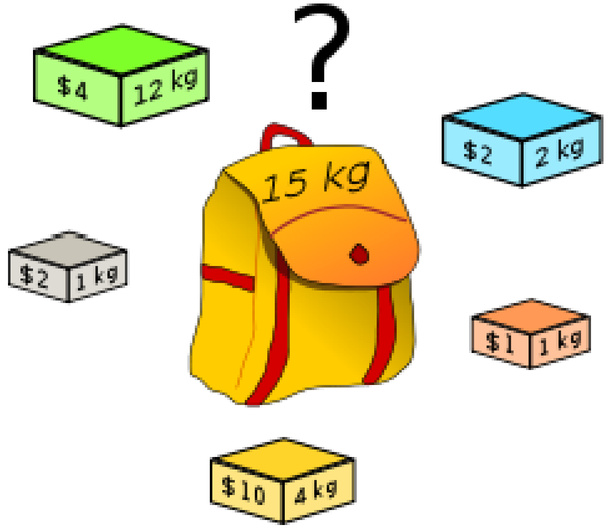
0 of 54

Can we use DP technique on top of the naïve solution to make it more efficient?

1. Yes, of course
2. No, that naïve solution does not have recurrence

0 of 54

0-1 Knapsack: DP Solution (1)



- Distinct states:
  - Instead of considering **all items** at a time let's consider **one item** at a time
    - Give **id** to each item, from 0 to N-1
    - Example: (4, 12), (2, 1), (10, 4), (2, 2), (1, 1)
  - For each item, we can take or ignore it
  - But this parameter **id** alone is not enough
  - We need another parameter:
    - The current weight **w_left**
  - Space Complexity: O(N * |W|)

0-1 Knapsack: DP Solution (2)

- Overlapping transitions:
  - knapsack(N, w_left) = 0 // all items have been considered
  - knapsack(id, 0) = 0 // we cannot carry anything else
  - knapsack(id, w_left) = max(
    knapsack(id + 1, w_left),
    // that is, we ignore this item id
    value[id] + knapsack(id + 1, w_left - weight[id]))
    // or take item id, but only if weight[id] <= w_left
  - Time Complexity = O(N * |W| * 1) = O(N * |W|)
    - This is called pseudo-polynomial
- See UVa10130.java (top-down implementation)
  - http://uva.onlinejudge.org/external/101/10130.html

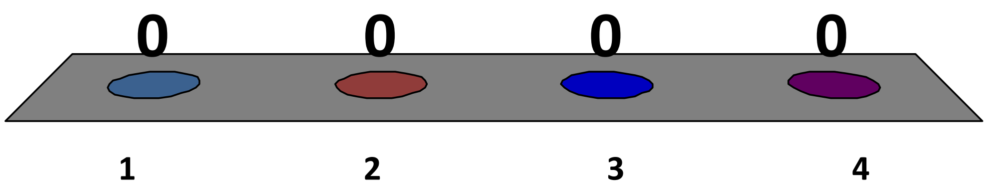
5 minutes break

These slides are originally from A/P Sung Wing Kin, Ken

DP ON STRING

Quick Survey: String Alignment

1. I have not seen this problem before, this is from a level 3 module?
2. I have seen this problem before
3. I have solved (coded a solution for) this problem before
4. On top of no 3, I also know its speed up techniques and variants



0 of 54

String Edit Problem (1)

- Given two strings A and B, edit A to B with the minimum number of **edit operations**:
  - Replace a letter with another letter
  - Insert a letter
  - Delete a letter
- e.g.
  - A = ACAATCC   A_CAATCC
  - B = AGCATGC   AGCA_TGC
                        01001010
  - Edit distance = 3, sum of all ‘1’ above

String Edit Problem (2)

- Instead of minimizing the number of edge operations, we can associate a **cost function** to the operations and minimize the total cost
  - Such cost is called **edit distance**
- For the previous example, the cost is as follows:
  - A = A_CAATCC
  - B = AGCA_TGC
    0100101 0
  - Edit distance = 3

	_	A	C	G	T
_		1	1	1	1
A	1	0	1	1	1
C	1	1	0	1	1
G	1	1	1	0	1
T	1	1	1	1	0

String Alignment Problem (1)

- Instead of using **string edit**, in computational biology, people like to use **string alignment**
- We use **similarity function**, instead of **cost function**, to evaluate the goodness of the alignment
  - e.g. we give 2 points for match; -1 point for mismatch, insert, or delete

$\delta(C,G) = -1$

	_	A	C	G	T
_		-1	-1	-1	-1
A	-1	2	-1	-1	-1
C	-1	-1	2	-1	-1
G	-1	-1	-1	2	-1
T	-1	-1	-1	-1	2

String Alignment Problem (2)

- Consider two strings **ACAATCC** and **AGCATGC**
- One of their **alignment** is

insert match

```
A_CAATCC
AGCA_TGC
```

delete mismatch

- In the above alignment,
  - Space ('_') is introduced to both strings
  - There are 5 matches, 1 mismatch, 1 insert, and 1 delete

String Alignment Problem (3)

- This alignment has similarity score 7

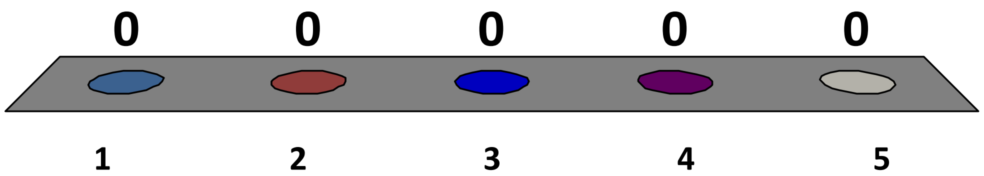
```
A_CAATCC
AGCA_TGC
```

- Note that the alignment above has maximum score
  - Such alignment is called the **optimal alignment**
- **String alignment problem** tries to find
  the alignment with the maximum similarity score!
- String alignment problem is also called
  the **global alignment problem**

To Test Your Understanding… What is the global alignment score of "STEVEN" and "SEVEN“?

2 points for match; -1 point for mismatch, insert, or delete

1. -1
2. 9
3. 5
4. 6
5. 7, of course 7!!
   This is a trick question!



0 of 54

Needleman-Wunsch DP Algorithm (1)

- Consider two strings S[1..n] and T[1..m]
- Define V(i, j) be the score of the optimal alignment between the prefixes S[1..i] and T[1..j]
- Base Cases:
  - $V(0, 0) = 0$
  - $V(0, j) = V(0, j-1) + \delta(\_, T[j])$ for $j \in [1..m]$
    - Insert j times
  - $V(i, 0) = V(i-1, 0) + \delta(S[i], \_)$ for $i \in [1..n]$
    - Delete i times

Needleman-Wunsch DP Algorithm (2)

- Recurrences: For i>0, j>0

$$V(i,j)=\max\begin{cases} V(i-1,j-1)+\delta(S[i],T[j]) & \text{Match/mismatch} \\ V(i-1,j)+\delta(S[i],\_) & \text{Delete} \\ V(i,j-1)+\delta(\_,T[j]) & \text{Insert} \end{cases}$$

- In the alignment, the last pair must be either match/mismatch, delete, insert

```
xxx…xx          xxx…xx        xxx…x_
     |               |             |
xxx…yy          yyy…y_        yyy…yy
```

match/mismatch   delete   insert

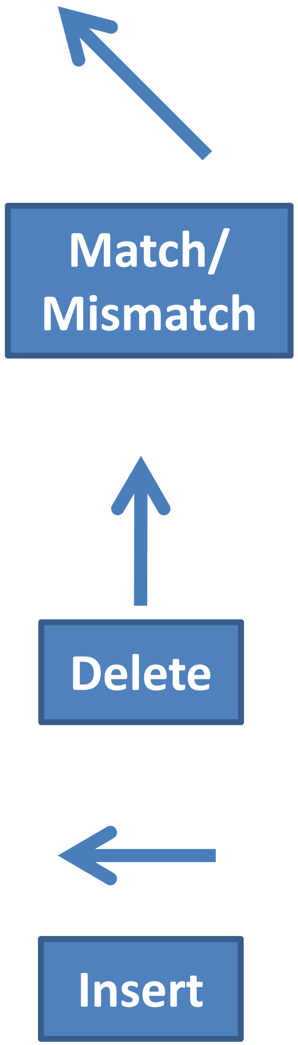
Example (1) – Base Cases

Usually implemented bottom-up

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1							
C	-2							
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

Example (2) – Recurrences

Because the topological order is easy: row-by-row, left-to-right



	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3	2			
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

What

What is the value of V(3,4)?

Example (3) – Complete DP Table

Can you find the implicit DAG?

No OP/ Replace

Delete

Insert

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3	2	1	0	-1
A	-3	0	0	2	5	4	3	2
A	-4	-1	-1	1	4	4	3	2
T	-5	-2	-2	0	3	6	5	4
C	-6	-3	-3	0	2	5	5	7
C	-7	-4	-4	-1	1	4	4	7

Analysis

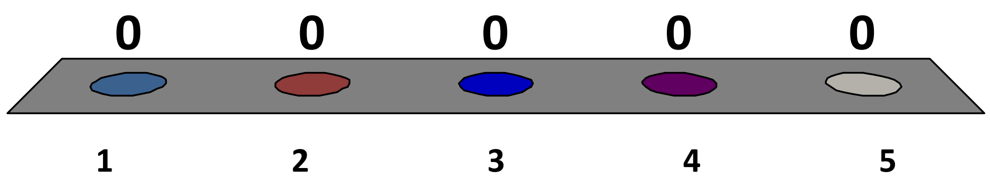
- Distinct States:
  We need to fill in all entries in an $n \times m$ matrix
- Space Complexity = $O(nm)$
- Overlapping Transitions: Each entries can be computed in $O(1)$ time, by looking at three other entries ☺
  - That clearly overlaps…
- Time Complexity = $O(nm * 1) = O(nm)$
- See StringAlignmentDemo.java

Overview of TUTOR

- What are the possible parameters of this problem?
  - Which subset of them are needed to get distinct states?
- What are the possible actions of this problem?
  - Can you write a recurrence based on these actions?
  - Are they cyclic?
  - Are they overlapping?

DP…

1. Looks very easy
2. Looks easy
3. Neutral
4. Looks hard
5. Looks very hard



0 of 54

Summary

- We have seen 3 examples of DP problems that are not natural to be seen as graph problems
- We can write recurrences on them and since those recurrences share overlapping subproblems, we can use DP technique
- We have seen two ways to implement DP recurrences, top-down and bottom-up
- We will see two more examples at recitation later