Algorithms

(Dynamic Programming)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

November 11, 2021

Contents

- GO Fibonacci Number
- GO Partitions
- GO Tiling Problem
- GO Count Paths in a Grid
- GO Binomial Coefficient
- GO Longest Common Substring ▷ Not a DP
  - GO Longest Common Subsequence
  - GO Edit Distance
  - GO Square Submatrix
- GO Knapsack Problem
  - GO Coin Change
  - GO Minimum Partition Difference
- GO Sieve of Eratosthenes
- GO Egg Dropping
- GO Floyd-Warshall's All-Pairs Shortest Path

Contents (continued)

- GO Matrix-Chain Multiplication
- GO Rod Cutting
- GO Staircase
- GO Jumps
- GO Card Game
- GO Wildcard Pattern Matching
- GO Coin Toss
- GO Dice Throw
- GO Painter's Partition
- GO Filling Bookcase Shelves
- GO Gold Mine
- GO Word Break
- GO Combinations
- GO Ugly Number
- GO Parenthesizations
- GO Derivations using Ambiguous Grammar
- GO Optimal Binary Search Tree

Contents (continued)

- GO Snakes and Ladders
- GO Cutting Cloth
- GO Arranging Balls
- GO Word Wrap
- GO Difficulty of Job Schedule
- GO Wine Sales
- GO Grid Traversals
- GO Sequence Alignment + Gaps

  GO Longest Palindromic Substring ▷ Not a DP

  GO Longest Palindromic Subsequence

  GO Longest Bitonic Subsequence

  GO Longest Repeating Subsequence

  GO Shortest Common Supersequence

  GO Longest Alternating Subsequence ▷ Not a DP

  GO Count Subsequence Occurrences

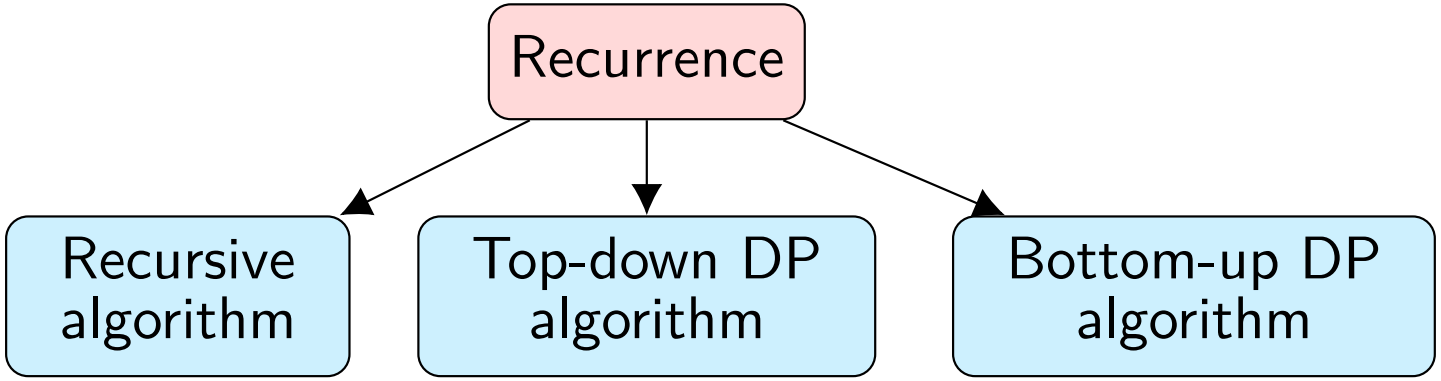
Contributors

- Angel Li
- Ahalya Sugumar
- Arnnav Prasad
- Manikanta Tadikonda
- Pratik Thorwe
- Namratha Kondapally
- Alekhya Kodali
- Sree Sai Nitish Gali
- Sneh Amit Shah
- Naman Banati
- Rakesh Tella
- Paras Malik
- Ayush Sharma

What is dynamic programming (DP)?

- DP is an algorithm design technique
- DP finds optimal solutions to a problem by combining optimal solutions to its overlapping subproblems by saving solutions to subproblems in a table and never recomputing them
- DP is efficient table filling where the value at each cell (solution to a subproblem) depends on the values at two or more other cells (solutions to overlapping subproblems)

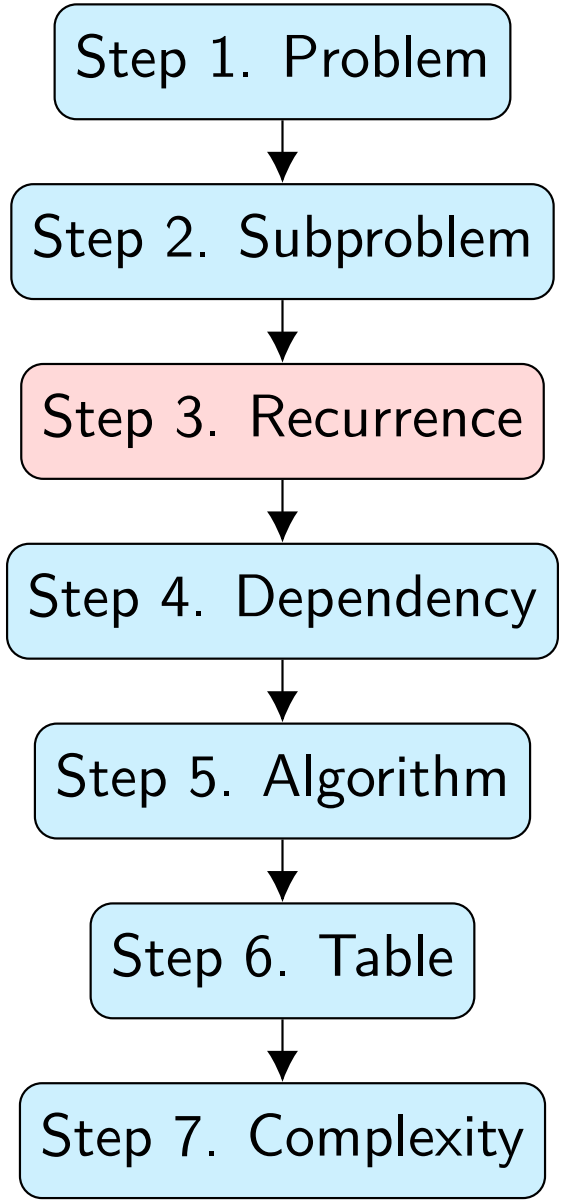
Recurrence leads to recursive/DP algorithms



- Recurrence is the brain of dynamic programming.
- Once we have a recurrence, it is relatively easy to derive a recursive or top-down DP or bottom-up DP algorithm.
- Bottom-up DP is typically the fastest in practice.

Feature	Recursive	Top-down DP	Bottom-up DP
Uses	Recursion	Memoization	Tabulation
Each subproblem computed	≥ 1 time	≤ 1 time	$= 1$ time
Computation graph	Tree	DAG	DAG
Coding	Very easy	Easy	Okay
Complexity	Exponential	Polynomial	Polynomial

DP problem-solving template



Fibonacci Number HOME

Step 1. Problem

- [Link] Compute the nth Fibonacci number

Step 2. Subproblem

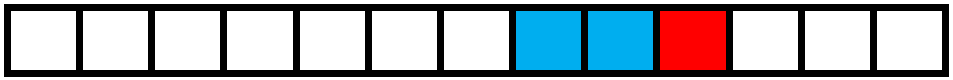
$$F[i] = i\text{th Fibonacci number}$$

Compute $F[n]$

Step 3. Recurrence

$$F[i] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, \\ 1 & \text{if } i = 1, \\ F[i-1] + F[i-2] & \text{if } i \in [2, n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

FIBONACCI(n)

Input: Whole number n
Output: Fibonacci number F_n

```
F[0] ← 0; F[1] ← 1
for i ← 2 to n do
  F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

Step 6. Table

i	0	1	2	3	4	5	6	7	8	9	10
$F[i]$	1	1	2	3	5	8	13	21	34	55	89

Step 7. Complexity

$$\text{Time} \in \Theta\left(n\right), \text{Space} \in \Theta\left(n\right)$$

Partitions HOME

Step 1. Problem

- [Link] Count the number of ways to express a number as a sum of 1, 3, and 4.
- Example: 4 can be expressed as a sum of 1, 3, and 4 in 4 ways:

$$4 = \begin{Bmatrix} 1+1+1+1 \\ 1+3 \\ 3+1 \\ 4 \end{Bmatrix}$$

Step 2. Subproblem

$$C[i] = \#\text{Ways of expressing } i \text{ as a sum of } 1, 3, 4$$

Compute $C[n]$

Step 3. Recurrence

$$C[i] = \left\{ \begin{array}{ll} 1 & \text{if } i = 1, \\ 1 & \text{if } i = 2, \\ 2 & \text{if } i = 3, \\ 4 & \text{if } i = 4, \\ C[i-1] + C[i-3] + C[i-4] & \text{if } i \in [5, n]. \end{array} \right\}$$

Step 4. Dependency

Step 5. Algorithm

$\textsc{NumberExpressedAsSum}(n)$

Input: Natural number n
Output: # Ways of expressing n as a sum of 1,3,4

```
C[1] ← 1; C[2] ← 1; C[3] ← 2; C[4] ← 4
for i ← 5 to n do
    C[i] ← C[i − 1] + C[i − 3] + C[i − 4]
return C[n]
```

Step 6. Table

i	1	2	3	4	5	6	7	8	9	10
$C[i]$	1	1	2	4	6	9	15	25	40	64

Step 7. Complexity

$$\text{Time} \in \Theta\left(n\right), \text{Space} \in \Theta\left(n\right)$$

Step 1. Problem

- [Link] For a given natural number n, find the number of different ways in which n can be written as a sum of one or more natural numbers.
- Example: Suppose $n = 5$. Then, there are 7 different ways to partition 5. They are: $\{5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1\}$.
- The terms $3+1+1$, $1+3+1$, and $1+1+3$ are considered the same because these are permutations of the same object and we disallow permutations of the same object.

Step 2. Subproblem

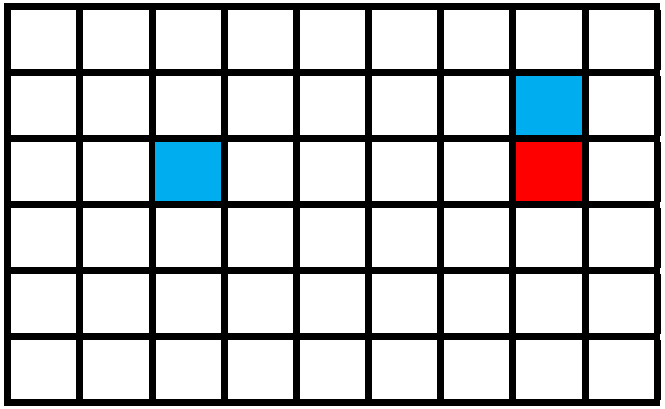
$P[i, j] =$ Partitions of natural number j using natural numbers less than or equal to natural number i.

Compute $P[n, n]$.

Step 3. Recurrence

$$P[i,j] = \left\{ \begin{array}{ll} 1 & \text{if } i \in [0,n], j = 0, \\ 0 & \text{if } i = 0, j \in [1,n], \\ P[i-1,j] + P[i,j-i] \times \boxed{j \geq i} & \text{if } i,j \in [1,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

Partitions(n)

Input: A natural number n.
Output: Partitions of n.

1. **for** $i \leftarrow 0$ **to** n **then** $P[i, 0] \leftarrow 1$
2. **for** $j \leftarrow 1$ **to** n **then** $P[0, j] \leftarrow 0$
3. **for** $i \leftarrow 1$ **to** n **do**
4.   **for** $j \leftarrow 1$ **to** $i - 1$ **do** $P[i, j] \leftarrow P[i - 1, j]$
5.   **for** $j \leftarrow i$ **to** n **do** $P[i, j] \leftarrow P[i - 1, j] + P[i, j - i]$
6. **return** $P[n, n]$

Step 6 Table

$P[i,j]$	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	3	3
3	1	1	2	3	4	5
4	1	1	2	3	5	6
5	1	1	2	3	5	7

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Tiling Problem HOME

Step 1. Problem

- [Link] Count the number of ways to tile a board of size $2 \times n$ using 1×2 or 2×1 tiles.

Step 2. Subproblem

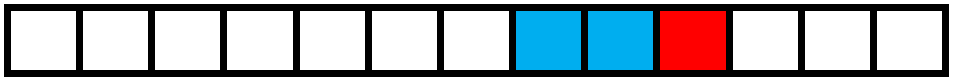
$C[i] = \#$Ways to tile a $2 \times i$ board using 1×2 or 2×1 tiles

Compute $C[n]$

Step 3. Recurrence

$$C[i] = \left\{ \begin{array}{ll} 1 & \text{if } i = 1, \\ 2 & \text{if } i = 2, \\ C[i-1] + C[i-2] & \text{if } i \in [3, n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

TILINGS(n)

Input: #Columns $n \geq 1$ (in the $2 \times n$ board)
Output: #Ways to tile the board using 1×2 or 2×1 tiles

```
C[1] ← 1; C[2] ← 2
for i ← 3 to n do
    C[i] ← C[i − 1] + C[i − 2]
return C[n]
```

Step 6. Table

i	1	2	3	4	5	6	7	8	9	10
$C[i]$	1	2	3	5	8	13	21	34	55	89

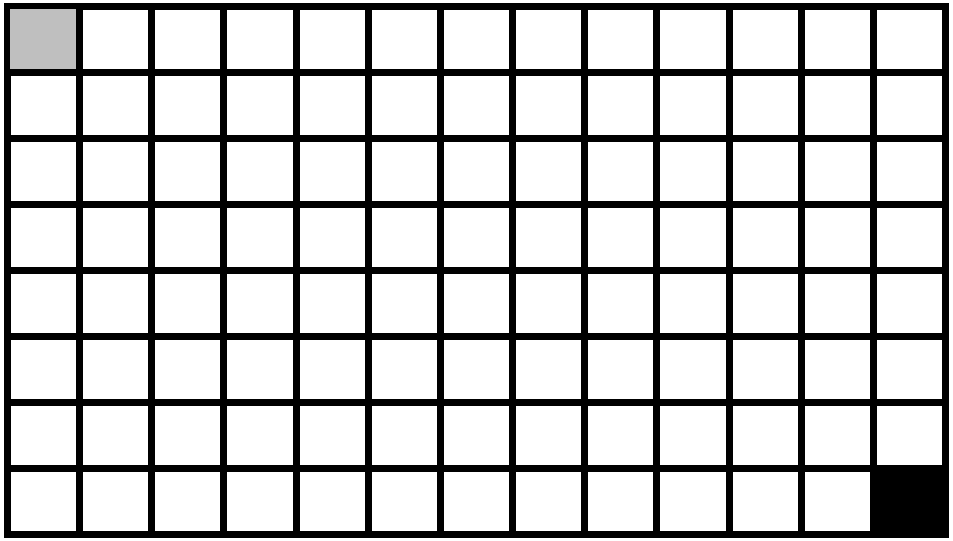
Step 7. Complexity

$$\text{Time} \in \Theta\left(n\right), \text{Space} \in \Theta\left(n\right)$$

Count Paths in a Grid HOME

Step 1. Problem

- [Link] Count all possible paths from top left corner [0, 0] to bottom right corner $[m, n]$ of a rectangular grid by only considering right and down moves.



Step 2. Subproblem

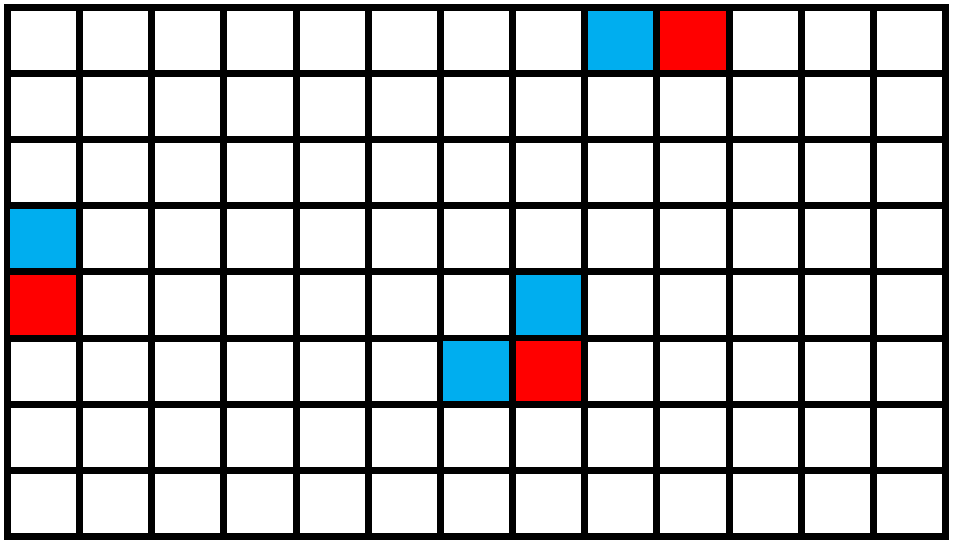
$$C[i,j] = \#\text{Paths to reach } [i,j] \text{ from } [0,0] \text{ with } \rightarrow, \downarrow$$

Compute $C[m,n]$

Step 3. Recurrence

$$C[i,j] = \left\{ \begin{array}{ll} 1 & \text{if } i = 0 \text{ and } j = 0, \\ \begin{pmatrix} C[i,j-1] \times \boxed{j \geq 1} + \\ C[i-1,j] \times \boxed{i \geq 1} \end{pmatrix} & \text{if } i \in [1,m] \text{ or } j \in [1,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{PathsOnAGrid}(m, n)$

Input: Number of rows m and number of columns n
Output: #Paths from top left $(0, 0)$ to bottom right (m, n)

```
for i ← 0 to m do C[i, 0] ← 1
for j ← 0 to n do C[0, j] ← 1
for i ← 1 to m do
  for j ← 1 to n do
    C[i, j] ← C[i − 1, j] + C[i, j − 1]
return C[m, n]
```

Step 6. Table

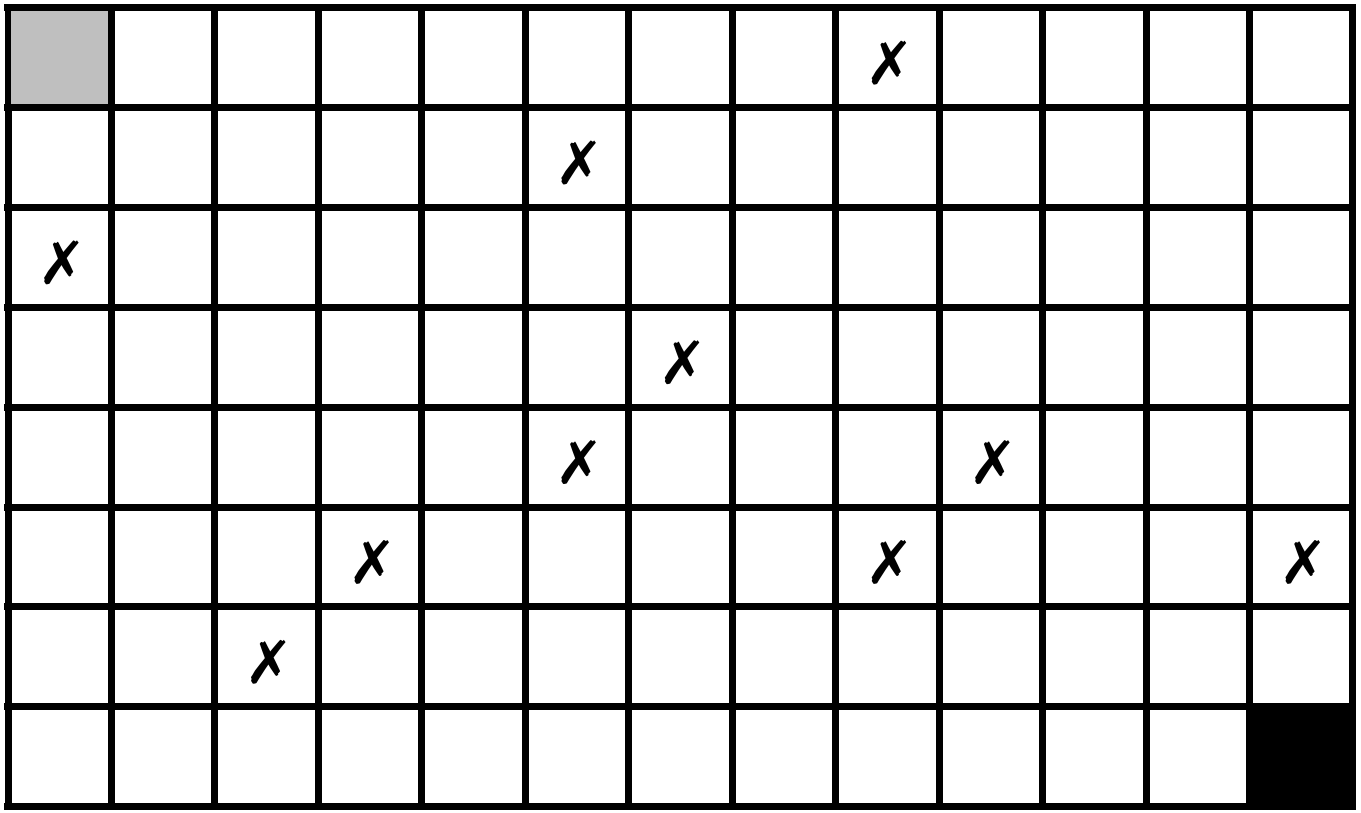
$C[i,j]$	0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	1	2	3	4	5	6	7	8
2	1	3	6	10	15	21	28	36
3	1	4	10	20	35	56	84	120
4	1	5	15	35	70	126	210	330
5	1	6	21	56	126	252	462	792
6	1	7	28	84	210	462	924	1716
7	1	8	36	120	330	792	1716	3432

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(mn\right)$$

Step 1. Problem

- [Link] Count all possible paths from top left corner $[0, 0]$ to bottom rightmost corner $[m, n]$ of a rectangular grid by only considering right and down moves.
- Cells with ✗ are blocked and cannot be passed.



Step 2. Subproblem

$$C[i,j] = \#\text{Paths to reach } [i,j] \text{ from } [0,0] \text{ with } \rightarrow, \uparrow.$$

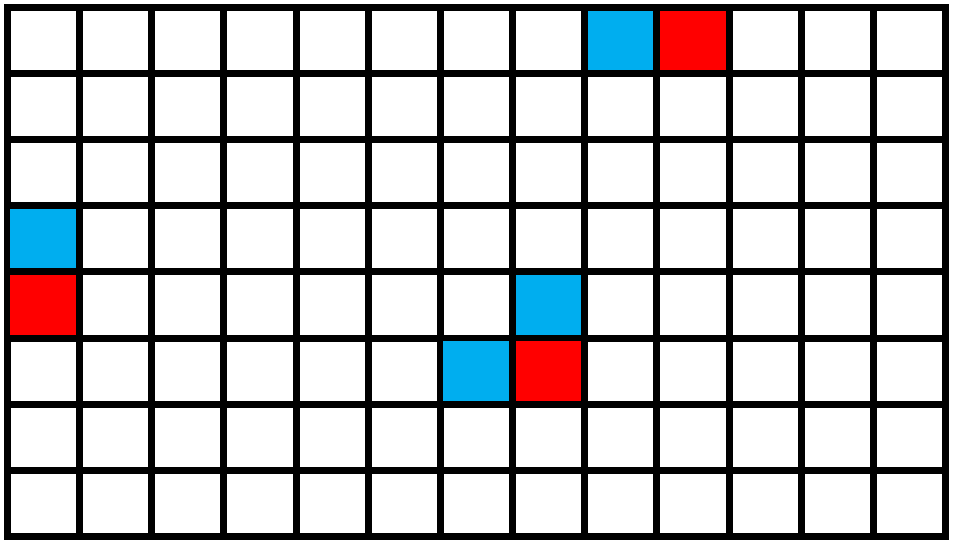
$$O[i,j] = \left\{ \begin{array}{ll} 1 & \text{if } [i,j] \text{ is open,} \\ 0 & \text{if } [i,j] \text{ is closed.} \end{array} \right\}$$

Compute $C[m,n]$.

Step 3. Recurrence

$$C[i,j] = \left\{ \begin{array}{ll} 1 \times O[i,j] & \text{if } i = 0, j = 0, \\ \left(\begin{array}{l} C[i,j-1] \times \boxed{j \geq 1} + \\ C[i-1,j] \times \boxed{i \geq 1} \end{array} \right) \times O[i,j] & \text{if } i \in [1,m] \text{ or } j \in [1,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{PathsOnAGridWithBlocks}(O[m,n])$

Input: Rectangular grid $O[m,n]$ with cells blocked denoted as 0 and open cells as 1.

Output: #Paths from top left $(0,0)$ to bottom right (m,n)

```
C[0,0] ← O[0,0]
for i ← 1 to m do C[i,0] ← C[i−1,0] × O[i,0]
for j ← 1 to n do C[0,j] ← C[0,j−1] × O[0,j]
for i ← 1 to m do
  for j ← 1 to n do
    C[i,j] ← (C[i−1,j] + C[i,j−1]) × O[i,j]
return C[m,n]
```

Step 6. Table

$O[i,j]$	0	1	2	3	4
0	1	1	0	1	1
1	1	1	1	0	0
2	1	1	1	1	1
3	0	1	0	1	1
4	1	1	1	1	1

$C[i,j]$	0	1	2	3	4
0	1	1	0	0	0
1	1	2	2	0	0
2	1	3	5	5	5
3	0	3	0	5	10
4	0	3	3	8	18

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(mn\right)$$

Binomial Coefficient HOME

Step 1. Problem

- [Link] Compute n choose r i.e., $C[n, r]$, where,

$$(x+y)^n = \sum_{i=0}^{n} C[n, i] x^{n-i} y^i$$

Step 2. Subproblem

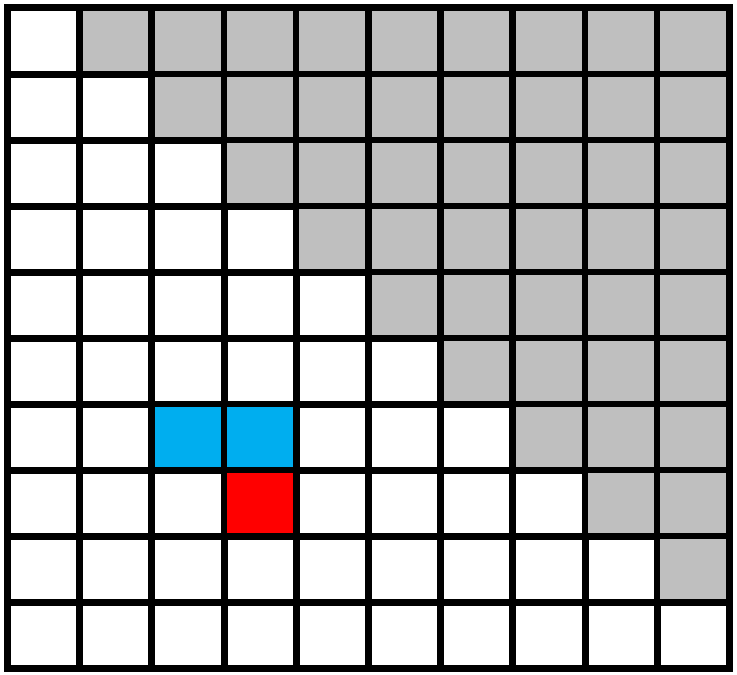
$C[i, j] = i$ choose j such that $0 \leq j \leq i$.

Compute $C[n, r]$.

Step 3. Recurrence

$$C[i,j] = \left\{ \begin{array}{ll} 1 & \text{if } j = 0 \text{ or } j = i, \\ C[i-1,j-1] + C[i-1,j] & \text{if } j \in [1, i-1]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{BinomialCoefficient}(n, r)$

Input: Two whole numbers n and r such that $n \geq r$
Output: Number of r-sized subsets from an n-element set

```
for i ← 0 to n do
  for j ← 0 to min(i, r) do
    if j = 0 or j = i do
      C[i, j] ← 1
    else
      C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
return C[n, r]
```

Step 6. Table

$C[i,j]$	0	1	2	3	4	5	6	7	8	9
0	1									
1	1	1								
2	1	2	1							
3	1	3	3	1						
4	1	4	6	4	1					
5	1	5	10	10	5	1				
6	1	6	15	20	15	6	1			
7	1	7	21	35	35	21	7	1		
8	1	8	28	56	70	56	28	8	1	
9	1	9	36	84	126	126	84	36	9	1

Step 7. Complexity

$$\text{Time} \in \Theta\left(nr\right), \text{Space} \in \Theta\left(nr\right)$$

Longest Common Substring HOME

Step 1. Problem

- [Link] Compute length of the longest common substring between two strings $x[1..m]$ and $y[1..n]$.

Step 2. Subproblem

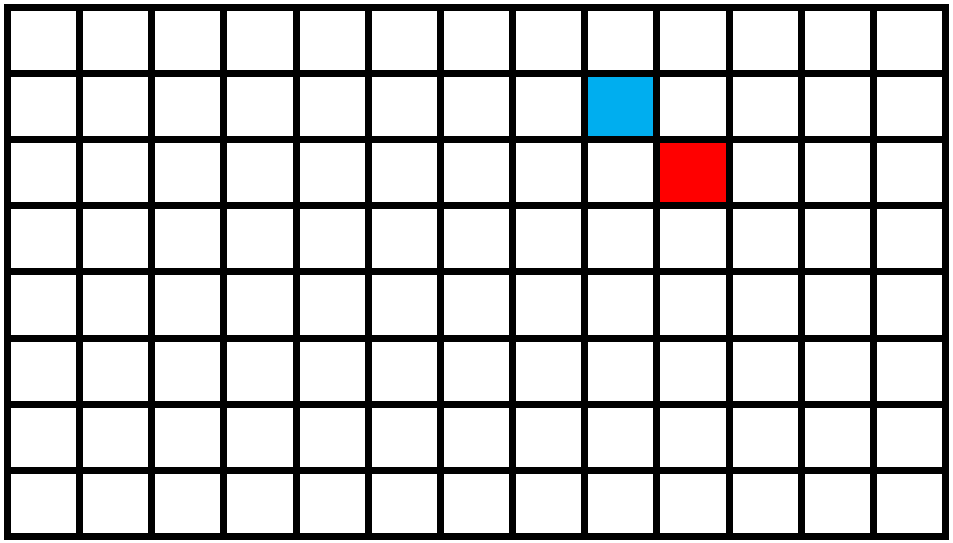
$L[i, j] =$ Length of the longest common suffix between strings $x[1..i]$ and $y[1..j]$.

Compute $\max(L[1..m, 1..n])$.

Step 3. Recurrence

$$L[i,j] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0 \text{ or } j = 0, \\ (L[i-1,j-1]+1) \times \boxed{x[i] = y[j]} & \text{if } i \in [1,m], j \in [1,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{LongestCommonSubstring}(x[1..m], y[1..n])$

Input: Strings $x[1..m]$, $y[1..n]$
Output: Length of the longest common substring between strings.

```
for i ← 0 to m do L[i, 0] ← 0
for j ← 1 to n do L[0, j] ← 0
for i ← 1 to m do
  for j ← 1 to n do
    if x[i] = y[j] then
      L[i, j] ← L[i − 1, j − 1] + 1
    else if x[i] ≠ y[j] then
      L[i, j] ← 0
return max(L[1..m, 1..n])
```

Step 6. Table

$L[i,j]$	0 ($y[0] = \varnothing$)	1 ($y[1] = n$)	2 ($y[2] = e$)	3 ($y[3] = w$)	4 ($y[4] = t$)	5 ($y[5] = o$)	6 ($y[6] = n$)
0 ($x[0] = \varnothing$)	0	0	0	0	0	0	0
1 ($x[1] = e$)	0	0	1	0	0	0	0
2 ($x[2] = i$)	0	0	0	0	0	0	0
3 ($x[3] = n$)	0	1	0	0	0	0	1
4 ($x[4] = s$)	0	0	0	0	0	0	0
5 ($x[5] = t$)	0	0	0	0	1	0	0
6 ($x[6] = e$)	0	0	1	0	0	0	0
7 ($x[7] = i$)	0	0	0	0	0	0	0
8 ($x[8] = n$)	0	1	0	0	0	0	1

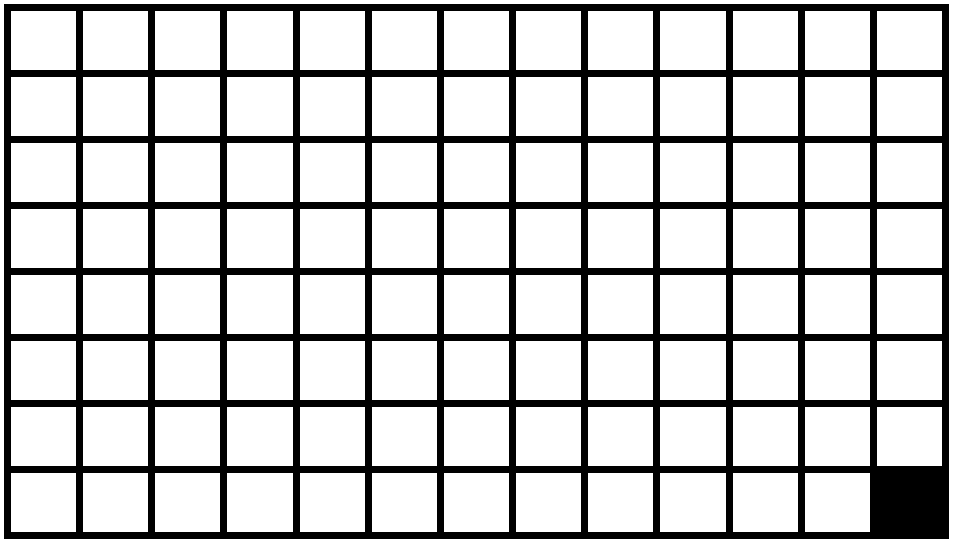
Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(mn\right)$$

Longest Common Subsequence HOME

Step 1. Problem

- [Link] Compute the length of the longest common subsequence (LCS) between two strings x and y of lengths m and n, respectively.



Step 2. Subproblem

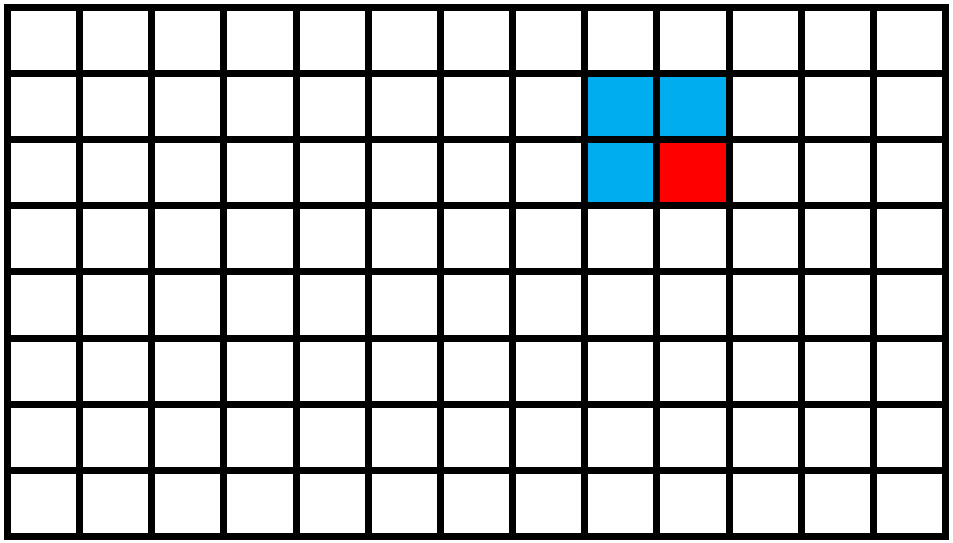
$L[i, j] =$ Length of the LCS between strings $x[1..i]$ and $y[1..j]$.

Compute $L[m, n]$.

Step 3. Recurrence

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \left\{ \begin{array}{l} (L[i-1,j-1]+1) \times \boxed{x[i] = y[j]} + \\ \max \left\{ \begin{array}{l} L[i,j-1], \\ L[i-1,j] \end{array} \right\} \times \boxed{x[i] \neq y[j]} \end{array} \right\} & \text{if } i \in [1,m], j \in [1,n]. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{LongestCommonSubsequence}(x[1..m], y[1..n])$

Input: Strings $x[1..m]$, $y[1..n]$.
Output: Length of the longest common subsequence between the strings.

```
for i ← 0 to m do L[i, 0] ← 0
for j ← 1 to n do L[0, j] ← 0
for i ← 1 to m do
  for j ← 1 to n do
    if x[i] = y[j] then
      L[i, j] ← L[i − 1, j − 1] + 1
    else if x[i] ≠ y[j] then
      L[i, j] ← max(L[i − 1, j], L[i, j − 1])
return L[m, n]
```

Step 6. Table

$L[i,j]$	0 ($y[0] = \varnothing$)	1 ($y[1] = n$)	2 ($y[2] = e$)	3 ($y[3] = w$)	4 ($y[4] = t$)	5 ($y[5] = o$)	6 ($y[6] = n$)
0 ($x[0] = \varnothing$)	0	0	0	0	0	0	0
1 ($x[1] = e$)	0	0	1	1	1	1	1
2 ($x[2] = i$)	0	0	1	1	1	1	1
3 ($x[3] = n$)	0	1	1	1	1	1	2
4 ($x[4] = s$)	0	1	1	1	1	1	2
5 ($x[5] = t$)	0	1	1	1	2	2	2
6 ($x[6] = e$)	0	1	2	2	2	2	2
7 ($x[7] = i$)	0	1	2	2	2	2	2
8 ($x[8] = n$)	0	1	2	2	2	2	3

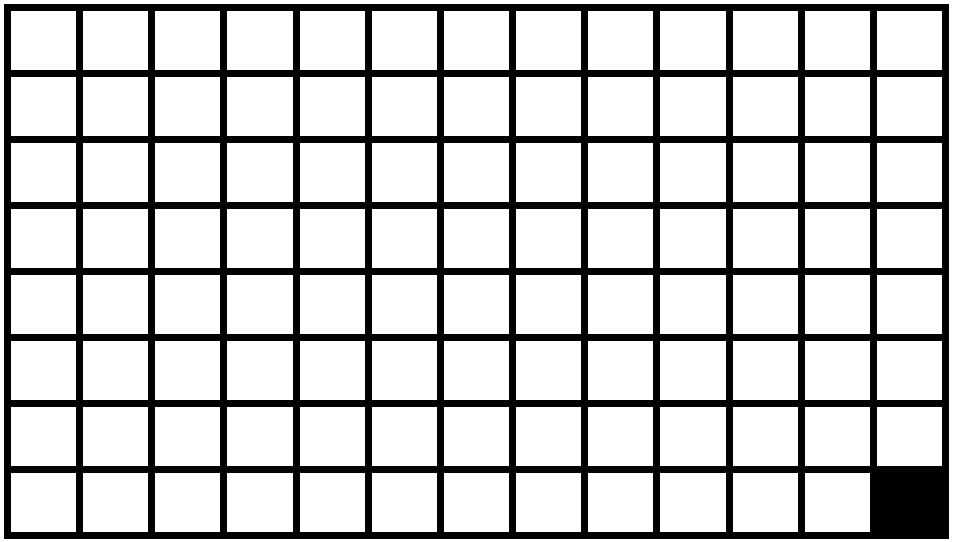
Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(mn\right)$$

Edit Distance HOME

Step 1. Problem

- [Link] Compute the minimum number of edits required to convert string $x[1..m]$ into string $y[1..n]$ using the three operations: insert, remove, and replace.



Step 2. Subproblem

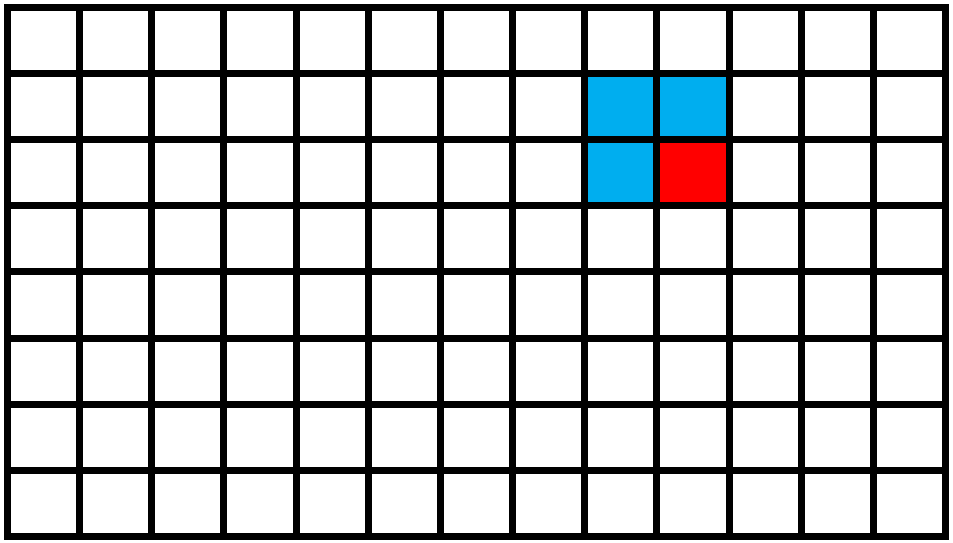
$D[i, j] =$ Least editing distance between strings $x[1..i]$ and $y[1..j]$.

Compute $D[m, n]$.

Step 3. Recurrence

$$D[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \left\{ \begin{array}{l} D[i-1,j-1] \times \boxed{x[i] = y[j]} + \\ \min \left\{ \begin{array}{l} D[i-1,j]+1, \\ D[i,j-1]+1, \\ D[i-1,j-1]+1 \end{array} \right\} \times \boxed{x[i] \neq y[j]} \end{array} \right\} & \text{if } i \in [1,m], j \in [1,n]. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{EditDistance}(x[1..m], y[1..n])$

Input: Strings $x[1..m]$, $y[1..n]$.
Output: Least editing distance between the strings.

```
for i ← 0 to m do D[i, 0] ← i
for j ← 1 to n do D[0, j] ← j
for i ← 1 to m do
  for j ← 1 to n do
    if x[i] = y[j] then
      D[i, j] ← D[i − 1, j − 1]
    else if x[i] ≠ y[j] then
      D[i, j] ← min(D[i − 1, j], D[i, j − 1], D[i − 1, j − 1]) + 1
return D[m, n]
```

Step 6. Table

$D[i,j]$	0 ($y[0] = \varnothing$)	1 ($y[1] = n$)	2 ($y[2] = e$)	3 ($y[3] = w$)	4 ($y[4] = t$)	5 ($y[5] = o$)	6 ($y[6] = n$)
0 ($x[0] = \varnothing$)	0	1	2	3	4	5	6
1 ($x[1] = e$)	1	1	1	2	3	4	5
2 ($x[2] = i$)	2	2	2	2	3	4	5
3 ($x[3] = n$)	3	2	3	3	3	4	4
4 ($x[4] = s$)	4	3	3	4	4	4	5
5 ($x[5] = t$)	5	4	4	4	4	5	5
6 ($x[6] = e$)	6	5	4	5	5	5	6
7 ($x[7] = i$)	7	6	5	5	6	6	6
8 ($x[8] = n$)	8	7	6	6	6	7	6

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(mn\right)$$

Square Submatrix HOME

Step 1. Problem

- [Link] Given a binary matrix $b[1..m, 1..n]$ containing 1's and 0's, find the maximum size of square submatrix with all 1's.

Step 2. Subproblem

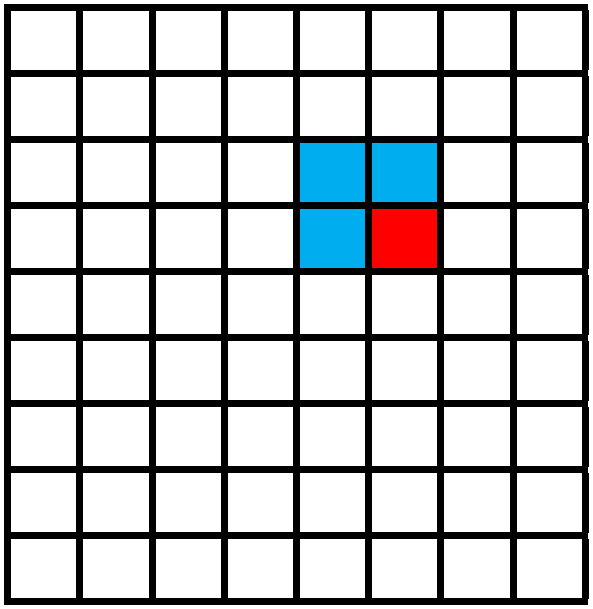
$M[i, j] =$ Maximum size of a square submatrix in $b[1..i, 1..j]$ such that $b[i, j]$ is the bottom-right cell of the submatrix.

Compute $\max(M[1..m, 1..n])$.

Step 3. Recurrence

$$M[i,j] = \left\{ \begin{array}{ll} \boxed{b[i,j]=1} & \text{if } i=1 \text{ or } j=1, \\ \boxed{b[i,j]=1} \times \left(1 + \min \begin{pmatrix} M[i-1,j], \\ M[i,j-1], \\ M[i-1,j-1] \end{pmatrix}\right) & \text{if } i \in [2,m], j \in [2,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{MaximumSquareSubmatrix}(b[1..m, 1..n])$

Input: Binary matrix $b[1..m, 1..n]$ containing 0's and 1's.
Output: Maximum size of a square submatrix containing 1's.

```
for i ← 1 to m do M[i, 1] = b[i, 1]
for j ← 2 to n do M[1, j] = b[1, j]
for i ← 2 to m do
  for j ← 2 to n do
    if b[i, j] = 0 then
      M[i, j] = 0
    else if b[i, j] = 1 then
      M[i, j] = 1 + min(M[i − 1, j], M[i, j − 1], M[i − 1, j − 1])
result ← 0
for i ← 1 to m do
  for j ← 1 to n do
    result = max(result, M[i, j])
return result
```

Step 6. Table

$b[i,j]$	1	2	3	4	5
1	1	0	1	0	0
2	1	0	1	1	1
3	1	1	1	1	1
4	1	0	0	1	0

$M[i,j]$	1	2	3	4	5
1	1	0	1	0	0
2	1	0	1	1	1
3	1	1	1	2	2
4	1	0	0	1	0

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(mn\right)$$

Knapsack Problem HOME

Step 1. Problem

- [Link] Given a set of n items numbered from 1 to n such that item i has weight $w[i]$ and value $v[i]$, find the maximum total value of the items that can fit in a knapsack with the maximum weight capacity of W in two cases:
  (i) [0-1.] each item can be considered only once.
  (ii) [Unbounded.] each item can be considered ∞ times.
- Let total items $n = 5$, weights $w[1..5] = [2, 3, 7, 9, 15]$, values $v[1..5] = [10, 15, 25, 13, 55]$, and capacity $W = 12$. Then
  (i) [0-1.] maximum total value $= 50$.
  (ii) [Unbounded.] maximum total value $= 60$.

Step 2. Subproblem

$K[i, j] =$ Maximum total value the knapsack can hold for the first i items with capacity at most j, where ith item's weight is $w[i]$ and value is $v[i]$

Compute $K[n, W]$

Step 3. Recurrence

- 0-1 knapsack

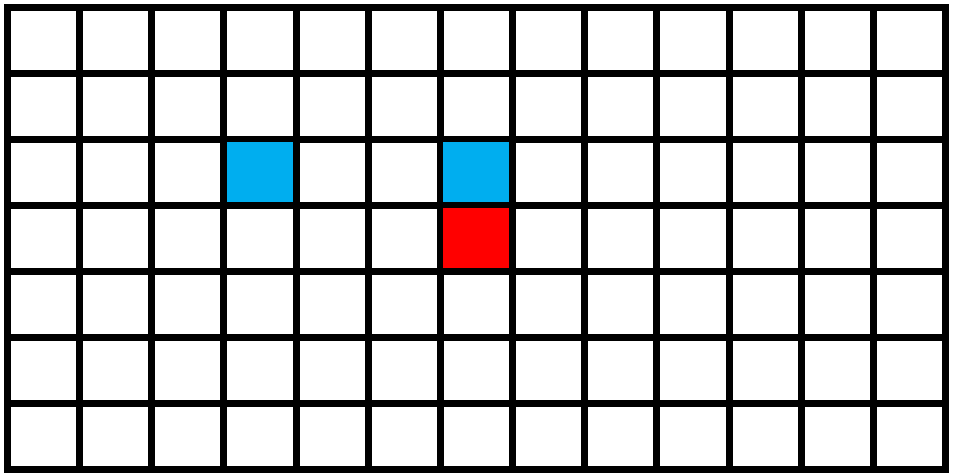
$$K[i,j] = \begin{cases} 0 & \text{if } ij = 0, \\ \max \begin{cases} K[i-1,j], \\ (K[i-1, j-w[i]] + v[i]) \times \boxed{j \geq w[i]} \end{cases} & \text{if } ij \geq 1. \end{cases}$$

- Unbounded knapsack

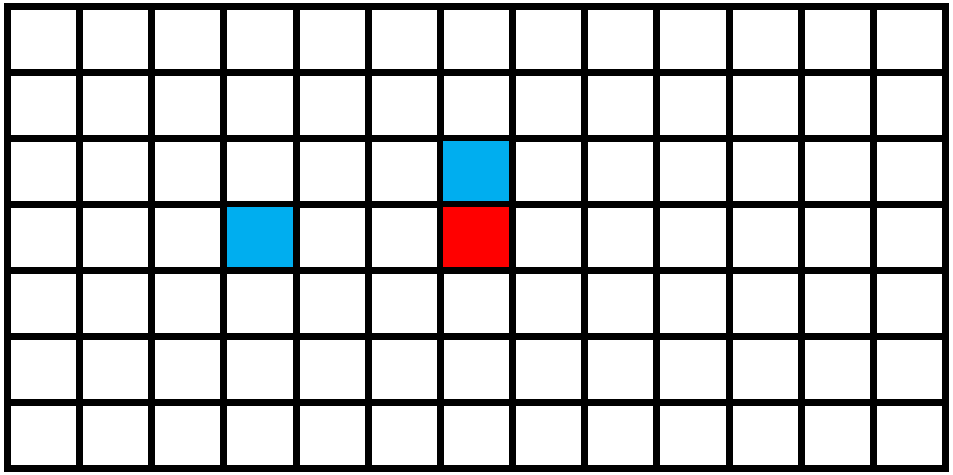
$$K[i,j] = \begin{cases} 0 & \text{if } ij = 0, \\ \max \begin{cases} K[i-1,j], \\ (K[i, j-w[i]] + v[i]) \times \boxed{j \geq w[i]} \end{cases} & \text{if } ij \geq 1. \end{cases}$$

Step 4. Dependency

- 0-1 knapsack



- Unbounded knapsack



Step 5. Algorithm

$\textsc{Knapsack}(w[1..n], v[1..n], W)$

Input: Items $i \in [1, n]$ with weight $w[i]$ and value $v[i]$
and knapsack's maximum weight capacity W
Output: Maximum total value of items that can fit in the knapsack
(This pseudocode works for both 0-1 and unbounded knapsack.
To use 0-1 knapsack, comment the unbounded knapsack recurrence.
Similarly, to use unbounded knapsack, comment the 0-1 knapsack recurrence.

```
K[0, 0..W] ← 0, K[0..n, 0] ← 0
for i ← 1 to n do
  for j ← 1 to W do
    if j ≥ w[i] then
      K[i, j] ← max (K[i − 1, j], K[i − 1, j − w[i]] + v[i])        ▷ 0-1
      K[i, j] ← max (K[i − 1, j], K[i, j − w[i]] + v[i])            ▷ unbounded
    else
      K[i, j] ← K[i − 1, j]
return K[n, W]
```

Step 6. Table

- 0-1 knapsack

$K[i,j]$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
$1\ (W[1]=2, V[1]=10)$	0	0	10	10	10	10	10	10	10	10	10	10	10
$2\ (W[2]=3, V[2]=15)$	0	0	10	15	15	25	25	25	25	25	25	25	25
$3\ (W[3]=7, V[3]=25)$	0	0	10	15	15	25	25	25	25	35	40	40	50
$4\ (W[4]=9, V[4]=13)$	0	0	10	15	15	25	25	25	25	35	40	40	50
$5\ (W[5]=15, V[5]=55)$	0	0	10	15	15	25	25	25	25	35	40	40	50

- Unbounded knapsack

$K[i,j]$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
$1\ (W[1]=2, V[1]=10)$	0	0	10	10	20	20	30	30	40	40	50	50	60
$2\ (W[2]=3, V[2]=15)$	0	0	10	15	20	25	30	35	40	45	50	55	60
$3\ (W[3]=7, V[3]=25)$	0	0	10	15	20	25	30	35	40	45	50	55	60
$4\ (W[4]=9, V[4]=13)$	0	0	10	15	20	25	30	35	40	45	50	55	60
$5\ (W[5]=15, V[5]=55)$	0	0	10	15	20	25	30	35	40	45	50	55	60

Step 7. Complexity

$$\text{Time} \in \Theta\left(nW\right), \text{Space} \in \Theta\left(nW\right)$$

Step 1. Problem

- Solve the unbounded knapsack problem using 1D space.

Step 2. Subproblem

$K[j]$ = Maximum total value the knapsack can hold with capacity at most j, for all given items, where ith item's weight is $w[i]$ and value is $v[i]$

Compute $K[W]$

Step 3. Recurrence

- Unbounded knapsack 1D

$$K[j] = \begin{cases} 0 & \text{if } j = 0 \\ \max \begin{Bmatrix} (K[j - w[1]] + v[1]) \times \boxed{j \geq w[1]}, \\ (K[j - w[2]] + v[2]) \times \boxed{j \geq w[2]}, \\ \vdots \\ (K[j - w[n]] + v[n]) \times \boxed{j \geq w[n]} \end{Bmatrix} & \text{if } j \in [1, W]. \end{cases}$$

Step 4. Dependency

Step 5. Algorithm

$\textsc{UnboundedKnapsack1D}(w[1..n], v[1..n], W)$

Input: A set of non-negative integers $w[1..n]$, a set of non-negative integers $v[1..n]$, and a non-negative integer value W

Output: Maximum total value the knapsack can hold with capacity at most W, for all given items where ith item's weight is $w[i]$ and value is $v[i]$

```
K[0..W] ← 0
for j ← 1 to W do
  for i ← 1 to n do
    if j ≥ w[i] then
      K[j] ← max(K[j], v[i]+K[j − w[i]])
return K[W]
```

Step 6. Table

$$w = [2, 3, 7, 9, 15],\ v = [10, 15, 25, 13, 55],\ W = 12$$

j	0	1	2	3	4	5	6	7	8	9	10	11	12
$K[j]$	0	0	10	15	20	25	30	35	40	45	50	55	60

Step 7. Complexity

$$\text{Time} \in \Theta\left(nW\right), \text{Space} \in \Theta\left(W\right)$$

Coin Change HOME

Step 1. Problem

- [Link] We have coins of denominations $c[1..m]$ such that $c[1] > c[2] > \cdots > c[m]$. Compute the minimum number of coins to make a change for n amount.
- Coin denominations $= \{9, 6, 5, 1\}$, $n = 11$, and mincoins $= 2$.

Step 2. Subproblem

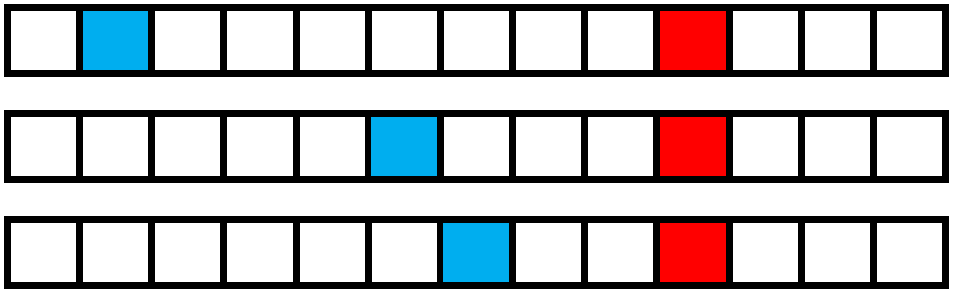
$M[i]$ = Minimum #coins to make a change for value i
using coin denominations $c[1] > c[2] > \cdots > c[m]$.

Compute $M[n]$.

Step 3. Recurrence

$$M[i] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, \\ \min \left\{ \begin{array}{c} (1 + M[i - c[1]]) \times \boxed{c[1] \leq i}, \\ (1 + M[i - c[2]]) \times \boxed{c[2] \leq i}, \\ \cdots \\ (1 + M[i - c[m]]) \times \boxed{c[m] \leq i} \end{array} \right\} & \text{if } i \geq 1. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{MinCoins}(c[1..m], n)$

Input: Coin denominations $c[1] > c[2] > \cdots > c[m]$ and amount n
Output: Minimum #coins to make change for n

```
M[0] ← 0
for i ← 1 to n do
  minimum ← ∞
  for j ← 1 to m do
    if c[j] ≤ i then
      minimum ← min(minimum, 1 + M[i − c[j]])
  M[i] ← minimum
return M[n]
```

Step 6. Table

$$n = 11$$

i	1	2	3	4
$c[i]$	9	6	5	1

i	0	1	2	3	4	5	6	7	8	9	10	11
$M[i]$	0	1	2	3	4	1	1	2	3	1	2	2

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(n\right)$$

Minimum Partition Difference HOME

Step 1. Problem

- [Link] Given a set of n integers $a[1..n]$, the task is to divide it into two partitions (can be multisets) such that the absolute difference between their sums is minimum. Compute this minimum absolute difference.
- For example, if the given input array is $[3, 15, 3, 2]$, then it can be partitioned optimally into two partitions $\{2, 3, 3\}$ and $\{15\}$ such that the absolute difference between the partition sums, i.e., $|(2 + 3 + 3) - (15)| = 7$ is the minimum possible absolute difference between any two partitions. Hence, the output is 7.

Step 2. Subproblem

$P[i, j] =$ Binary value to represent the existence of a subset in $a[0..i]$ that has sum j, where $a[0] = 0$.

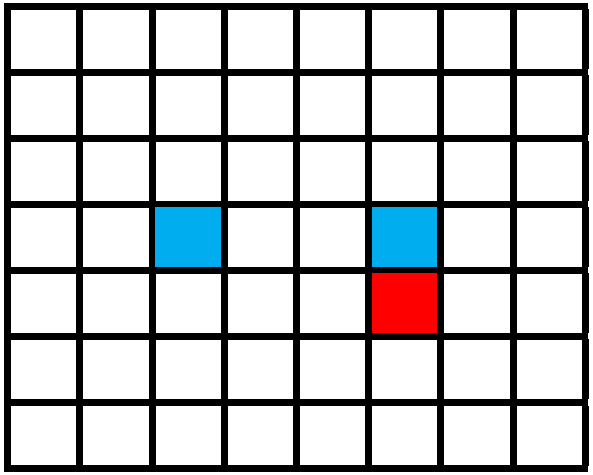
Suppose $s = a[1] + a[2] + \cdots + a[n]$.
Compute $\min_{j \in [0, s/2] \text{ and } P[n,j]=1} (s - 2j)$.

Step 3. Recurrence

Suppose $s = a[1] + a[2] + \cdots + a[n]$.

$$P[i,j] = \begin{cases} 1 & \text{if } i \in [0,n], j = 0, \\ 0 & \text{if } i = 0, j \in [1, s/2], \\ \left\{ \begin{array}{l} P[i-1,j] \text{ or} \\ P[i-1, j-a[i]] \times \boxed{j \geq a[i]} \end{array} \right\} & \text{if } i \in [1,n], j \in [1, s/2]. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{MinimumPartitionDifference}(a[1..n])$

Input: A set of non-negative integers $a[1..n]$, where $a[0] = 0$.
Output: Minimum absolute difference between two partitions of array a.

```
s ← 0
for i ← 1 to n do s ← s + a[i]
for i ← 0 to n do P[i, 0] ← 1
for j ← 1 to s/2 do P[0, j] ← 0
for i ← 1 to n do
  for j ← 1 to s/2 do
    P[i, j] ← P[i − 1, j]
    if j ≥ a[i] then
      P[i, j] ← P[i, j] or P[i − 1, j − a[i]]
result ← ∞
for j ← s/2 to 0 do
  if P[n, j] = 1 then
    result ← s − 2j; break
return result
```

Step 6. Table

$$s = 3 + 15 + 3 + 2 = 23.\ s/2 = 11.$$

$P[i,j]$	0	1	2	3	4	5	6	7	8	9	10	11
0 ($a[0] = 0$)	1	0	0	0	0	0	0	0	0	0	0	0
1 ($a[1] = 3$)	1	0	0	1	0	0	0	0	0	0	0	0
2 ($a[2] = 15$)	1	0	0	1	0	0	0	0	0	0	0	0
3 ($a[3] = 3$)	1	0	0	1	0	0	1	0	0	0	0	0
4 ($a[4] = 2$)	1	0	1	1	0	1	1	0	1	0	0	0

$$\text{Result} = s - 2 \times 8 = 23 - 16 = 7.$$

Step 7. Complexity

$$\text{Time} \in \Theta\left(nS\right), \text{Space} \in \Theta\left(nS\right)$$

Sieve of Eratosthenes HOME

Step 1. Problem

- [Link] Given a number n, find all prime numbers smaller than or equal to n. Suppose $n = 12$, then the prime numbers smaller than or equal to 12 are $2, 3, 5, 7, 11$.

Step 2. Subproblem

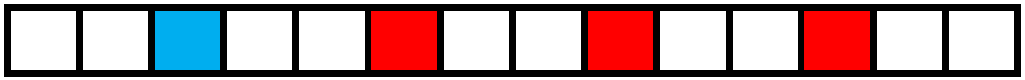
$P[i]$ = Boolean value indicating if number i is a prime or not.

Compute $P[2..n]$.

Step 3. Recurrence

$$P[i] = \left\{ \begin{array}{ll} true & \text{if } i = 2, \\ true & \text{if } i = 3, \\ \text{AND}_{j \in [2, \sqrt{i}] \text{ and } k \in [2, i \div j]} (P[j] \times \boxed{i \neq k \times j}) & \text{if } i \in [4, n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

SieveOfEratosthenes(n)

Input: The number n until which all prime numbers are to be found
Output: Boolean array $P[2..n]$ indicating prime numbers in the range $[2..n]$

```
for i ← 2 to n do
  P[i] ← true
for j ← 2 to ⌊√n⌋ do
  if P[j] = true then
    for k ← 2 to ⌊n/j⌋ do
      i ← j × k
      P[i] ← false
return P[2..n]
```

Step 6. Table

i	2	3	4	5	6	7	8	9	10	11	12
$P[i]$	T	T	F	T	F	T	F	F	F	T	F

Step 7. Complexity

$$\text{Time} \in \Theta\left(n \log \log n\right), \text{Space} \in \Theta\left(n\right)$$

Egg Dropping HOME

Step 1. Problem

- [Link] There is an n-floored building and we are given k identical eggs. A threshold floor of a building is defined as the highest floor in the building from and below which when the egg is dropped, the egg does not break, and above which when the egg is dropped, the egg breaks. Find the minimum number of drops required to find the threshold floor.

Step 2. Subproblem

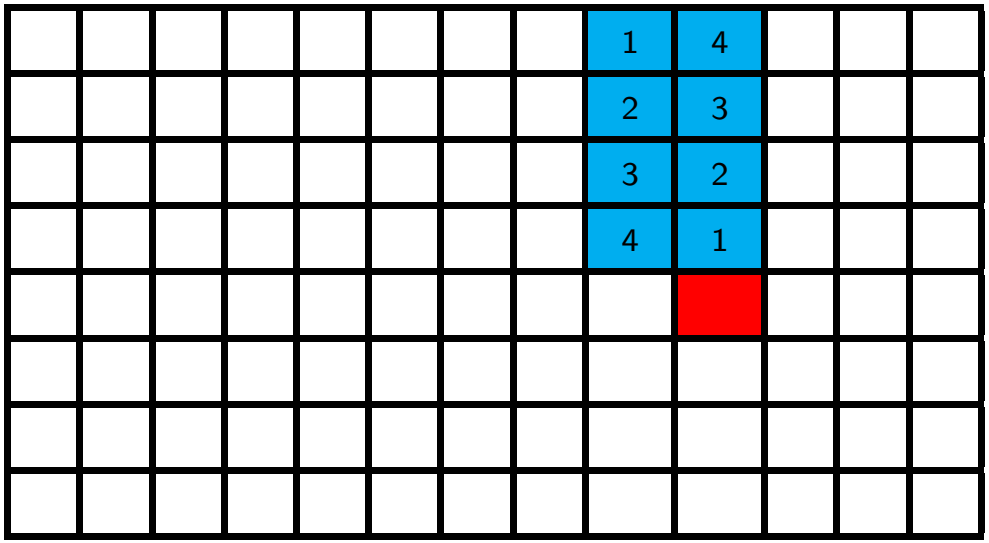
$D[i, j] = \#$Minimum drops to find the threshold floor in a building of i floors using j eggs.

Compute $D[n, k]$.

Step 3. Recurrence

$$D[i,j] = \left\{ \begin{array}{ll} i & \text{if } i \in [0,n], j = 1, \\ i & \text{if } i \in [0,1], j \in [2,k], \\ 1 + \min_{x \in [1,i]} \left\{ \max \left\{ \begin{array}{c} D[x-1, j-1], \\ D[i-x, j] \end{array} \right\} \right\} & \text{if } i \in [2,n], j \in [2,k]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

EggProblemMinimizeDrops(n, k)

Input: Number of floors n and number of eggs k
Output: Minimum #drops $D[n, k]$

```
for i ← 1 to n do
  D[i, 1] ← i
for j ← 1 to k do
  D[0, j] ← 0; D[1, j] ← 1
for i ← 2 to n do
  for j ← 2 to k do
    minimum ← i
    for x ← 1 to i do
      maximum ← max(D[x − 1, j − 1], D[i − x, j])
      if maximum < minimum then minimum ← maximum
    D[n, k] ← minimum + 1
return D[n, k]
```

Step 6. Table

$D[i,j]$	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2
3	3	2	2	2	2	2	2	2	2	2
4	4	3	3	3	3	3	3	3	3	3
5	5	3	3	3	3	3	3	3	3	3
6	6	3	3	3	3	3	3	3	3	3
7	7	4	3	3	3	3	3	3	3	3
8	8	4	4	4	4	4	4	4	4	4
9	9	4	4	4	4	4	4	4	4	4
10	10	4	4	4	4	4	4	4	4	4
11	11	5	4	4	4	4	4	4	4	4

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2 k\right), \text{Space} \in \Theta\left(nk\right)$$

Floyd Warshall's All-Pairs Shortest Path

HOME

Step 1. Problem

- [Link] We are given a directed weighted graph with n nodes numbered from 1 to n. The graph is represented as an adjacency matrix $w[1..n, 1..n]$, where $w[i, j]$ represents the weight of the directed edge from i to j. Find the length of the shortest path between every pair of vertices in the given graph.

Step 2. Subproblem

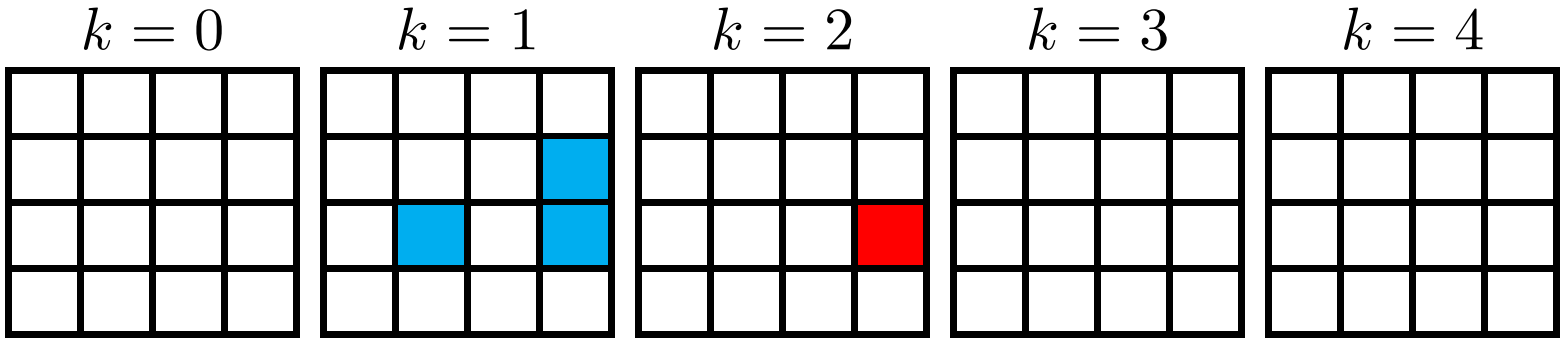
$D[k, i, j] =$ Length of the shortest path from node i to node j routing through nodes $\{1, 2, \ldots, k\}$.

Compute $D[n, 1..n, 1..n]$

Step 3. Recurrence

$$D[k,i,j] = \left\{ \begin{array}{ll} w[i,j] & \text{if } k = 0, \\ \min \left\{ \begin{array}{l} D[k-1,i,j], \\ D[k-1,i,k] + D[k-1,k,j] \end{array} \right\} & \text{if } k \in [1,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{FloydWarshallAPSP}(w[1..n, 1..n])$

Input: Adjacency matrix w of a weighted graph.
Output: Shortest distance matrix D.

```
D[0, 1..n, 1..n] ← w[1..n, 1..n]
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      D[k, i, j] ← min(D[k − 1, i, j], D[k − 1, i, k] + D[k − 1, k, j])
return D[n, 1..n, 1..n]
```

can be improved to

$\textsc{FloydWarshallAPSP}(w[1..n, 1..n])$

Input: Adjacency matrix w of a weighted graph.
Output: Shortest distance matrix D.

```
D[1..n, 1..n] ← w[1..n, 1..n]
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      D[i, j] ← min(D[i, j], D[i, k] + D[k, j])
return D[1..n, 1..n]
```

Step 6. Table

$D[0,i,j]$	1	2	3	4
1	0	∞	-2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	-1	∞	0

$D[1,i,j]$	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	∞	-1	∞	0

$D[2,i,j]$	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	3	-1	1	0

$D[3,i,j]$	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	4
3	∞	∞	0	2
4	3	-1	1	0

$D[4,i,j]$	1	2	3	4
1	0	-1	-2	∞
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^3\right), \text{Space} \in \Theta\left(n^2\right)$$

Matrix-Chain Multiplication HOME

Step 1. Problem

- [Link] Given a chain of n matrices $a_1, a_2, \ldots, a_n$ such that matrix i has dimension $p[i-1] \times p[i]$, find the minimum number of multiplications required to fully parenthesize the matrix-chain product $a_1 \times a_2 \times \cdots \times a_n$.

Step 2. Subproblem

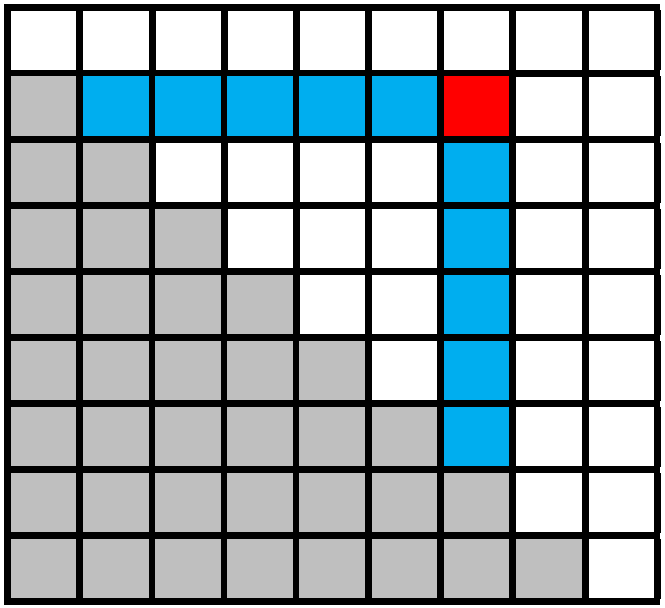
$M[i,j] =$ Minimum number of multiplications needed to compute the matrix-chain product $a_i \times a_{i+1} \times \cdots \times a_j$, where, the size of matrix a_i is $p[i-1] \times p[i]$.

Compute $M[1,n]$.

Step 3. Recurrence

$$M[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min \left\{ \begin{array}{l} M[i,i] + M[i+1,j] + p[i-1]p[i]p[j], \\ M[i,i+1] + M[i+2,j] + p[i-1]p[i+1]p[j], \\ \dots \\ M[i,j-1] + M[j,j] + p[i-1]p[j-1]p[j], \end{array} \right\} & \text{if } i < j. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{MatrixChainMultiplication}(p[0..n])$

Input: The size array $p[0..n]$
Output: Minimum #multiplications needed to multiply the matrix chain $a_1 \times a_2 \times \cdots \times a_n$ such that $p[i-1] \times p[i]$ is the size of matrix a_i.

```
for i ← 1 to n do
  M[i, i] ← 0
for ℓ ← 2 to n − 1 do                                  ▷ ℓ = chain length
  for i ← 1 to n − ℓ + 1 do
    j ← i + ℓ − 1
    M[i, j] ← ∞
    for k ← i to j − 1 do
      M[i, j] ← min(M[i, j], M[i, k] + M[k + 1, j] + (p[i − 1]p[k]p[j]))
return M[1, n]
```

Step 6. Table

i	0	1	2	3	4
$p[i]$	1	2	3	4	3

$M[i,j]$	1	2	3	4
1	0	6	18	30
2		0	24	48
3			0	36
4				0

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^3\right), \text{Space} \in \Theta\left(n^2\right)$$

Rod Cutting HOME

Step 1. Problem

- [Link] Compute maximum amount obtainable from cutting a rod of length n. We are given a list of prices $p[i]$ for $i \in [1, n]$ where $p[i]$ denotes the selling price of a rod of length i.

Step 2. Subproblem

$p[i] =$ Selling price of a rod of length i for $i \in [1, n]$

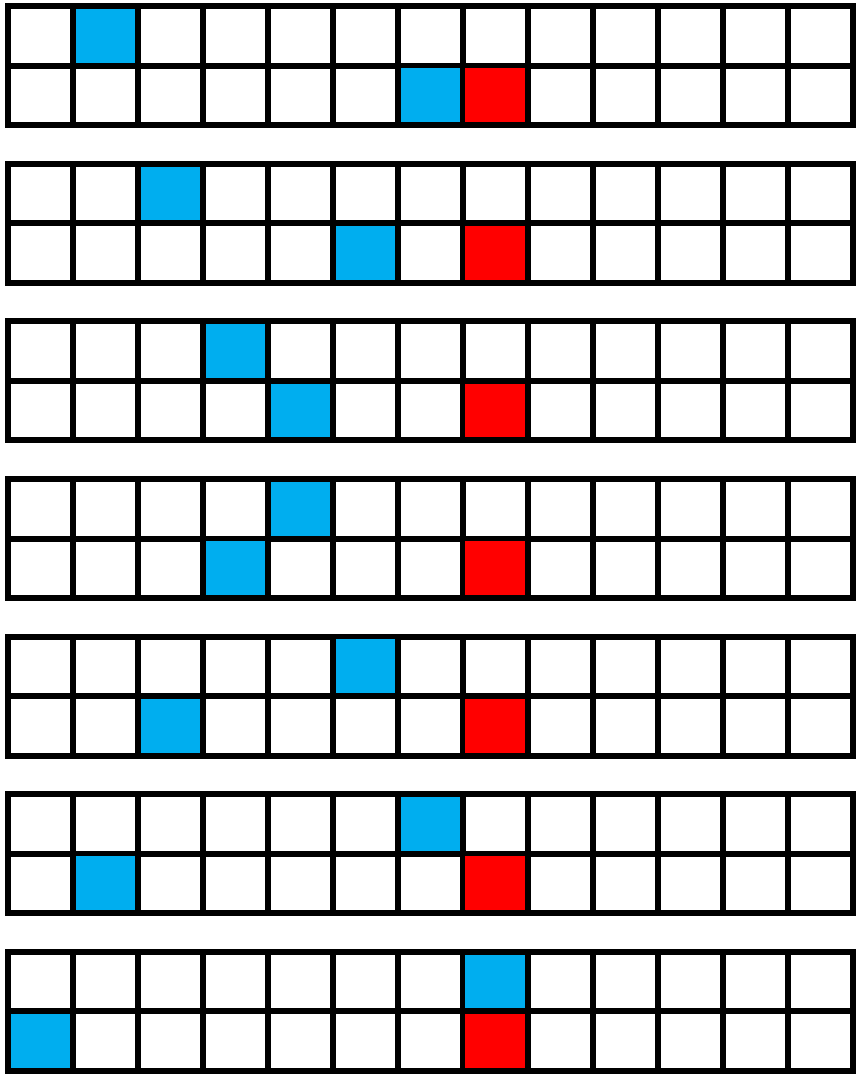
$R[i] =$ Revenue obtained from cutting a rod of length i where $i \in [0, n]$

Compute $R[n]$.

Step 3. Recurrence

$$R[i] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, \\ \max \left\{ \begin{array}{c} p[1] + R[i-1], \\ p[2] + R[i-2], \\ \vdots \\ p[i] + R[0] \end{array} \right\} & \text{if } i \in [1, n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{RodCutting}(p[1..n])$

Input: $p[1..n]$ where $p[i]$ is the selling price of a rod of length i.
Output: Maximum revenue obtainable from cutting a rod of length n

```
R[0] ← 0
for i ← 1 to n do
  R[i] ← −∞
  for j ← 1 to i do
    R[i] ← max(R[i], p[j] + R[i − j])
return R[n]
```

Step 6. Table

i	0	1	2	3	4	5	6
$p[i]$	$-$	1	5	8	17	17	20
$R[i]$	0	1	5	8	17	18	22

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n\right)$$

Staircase HOME

Step 1. Problem

- [Link] Count the number of ways of ascending a staircase of n steps assuming that a person can climb at most n steps in a single jump.

Step 2. Subproblem

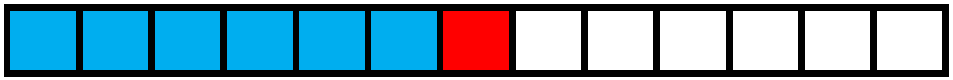
$C[i] = \#$Ways of ascending staircase of i steps.

Compute $W[n]$

Step 3. Recurrence

$$C[i] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, \\ 1 + C[0] + C[1] + \cdots + C[i-1] & \text{if } i \in [1, n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

StaircaseWays(n)

Input: The number of steps n.
Output: #Ways of ascending staircase of n steps.

```
C[0] ← 0
for i ← 1 to n do
  C[i] ← 1
  for j ← 0 to i − 1 do
    C[i] ← C[i] + C[j]
return C[n]
```

Step 6. Table

i	0	1	2	3	4	5	6	7	8
$C[i]$	0	1	2	4	8	16	32	64	128

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n\right)$$

Step 1. Problem

- [Link] Count the number of ways of ascending a staircase of n steps assuming that a person can climb at most $m \geq 1$ steps in a single jump.

Step 2. Subproblem

$C[i] = \#$Ways of ascending staircase of i steps
with at most m-sized jump, where $m \geq 1$.

Compute $W[n]$

Step 3. Recurrence

$$C[i] = \left\{ \begin{array}{ll} 1 & \text{if } i = 0, \\ C[0] + C[1] + \cdots + C[i-1] & \text{if } i \in [1, m], \\ C[i-m] + C[i-m+1] + \cdots + C[i-1] & \text{if } i \in [m, n]. \end{array} \right\}$$

Step 4. Dependency

Step 5. Algorithm

$\textsc{StaircaseWays}(n)$

Input: The number of steps n and at most m steps climbed in a single jump.
Output: #Ways of ascending n steps.

```
C[0] ← 1
for i ← 1 to m do
  C[i] ← 0
  for j ← 0 to i − 1 do
    C[i] ← C[i] + C[j]
for i ← m + 1 to n do
  C[i] ← 0
  for j ← i − m to i − 1 do
    C[i] ← C[i] + C[j]
return C[n]
```

Step 6. Table

i	0	1	2	3	4	5	6	7	8
$C[i]$ $(m=1)$	0	1	1	1	1	1	1	1	1
$C[i]$ $(m=2)$	0	1	2	3	5	8	13	21	34
$C[i]$ $(m=3)$	0	1	2	4	7	13	24	44	81
$C[i]$ $(m=4)$	0	1	2	4	8	15	29	56	108
$C[i]$ $(m=5)$	0	1	2	4	8	16	31	61	120
$C[i]$ $(m=6)$	0	1	2	4	8	16	32	63	125
$C[i]$ $(m=7)$	0	1	2	4	8	16	32	64	127
$C[i]$ $(m=8)$	0	1	2	4	8	16	32	64	128

Step 7. Complexity

$$\text{Time} \in \Theta\left(nm\right), \text{Space} \in \Theta\left(n\right)$$

Jumps HOME

Step 1. Problem

- [Link] Given an array of integers $a[1..n]$ where each element represents the maximum number of steps that can be made forward from that element. If an element is 0, we cannot move through that element.
- Find the minimum number of jumps to reach the end of the array, starting from the first element.

Step 2. Subproblem

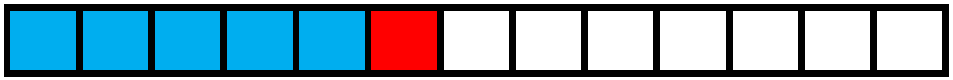
$J[i]$ = Minimum number of jumps to reach the ith element from the first element.

Compute $J[n]$.

Step 3. Recurrence

$$J[i] = \begin{cases} 0 & \text{if } i = 1, \\ \min \begin{cases} (J[1]+1) \times \boxed{(1+a[1]) \geq i}, \\ (J[2]+1) \times \boxed{(2+a[2]) \geq i}, \\ \dots \\ (J[i-1]+1) \times \boxed{(i-1+a[i-1]) \geq i} \end{cases} & \text{if } i \in [2, n]. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{MinimumJumps}(a[1..n])$

Input: Array $a[1..n]$
Output: Minimum #jumps $J[n]$ to reach the end of the given array

```
if n = 0 or a[1] = 0 return ∞
J[1] ← 0
for i ← 2 to n do
    J[i] ← ∞
    for j ← 1 to i do
        if j + a[j] ≥ i then
            J[i] ← min(J[i], J[j] + 1)
return J[n]
```

Step 6. Table

$$a[1..n] = [2, 1, 3, 0, 4, 2, 2, 1, 0, 1, 2, 3]$$

i	1	2	3	4	5	6	7	8	9	10	11	12
$J[i]$	0	1	1	2	2	2	3	3	3	∞	∞	∞

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n\right)$$

Card Game HOME

Step 1. Problem

- [Link] Two players play the following card game in turns, starting with player 1. There is a row of n cards facing up with values $c[0..n-1]$. Each player in his/her turn can pick either the first or the last card of the sequence. Total points of a player is the total value of all cards picked by that player. Find the maximum points player 1 can make.

Step 2. Subproblem

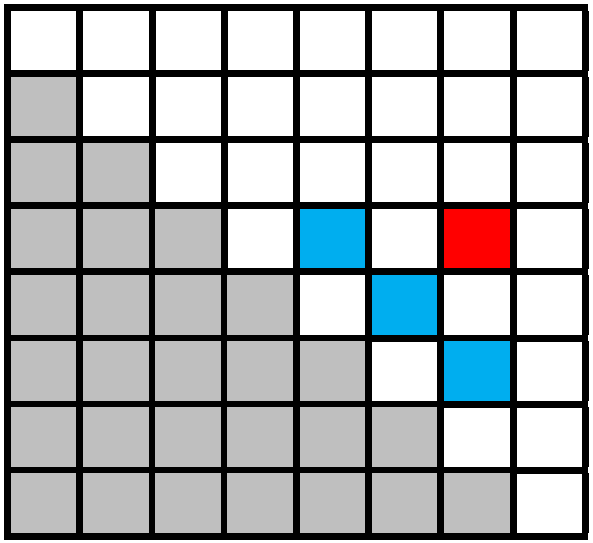
$M[i, j] =$ Maximum points that can be made by player 1 by selecting cards in the range $[i, j]$.

Compute $M[0, n-1]$.

Step 3. Recurrence

$$M[i,j] = \left\{ \begin{array}{ll} 0 & \text{if } i > j, \\ c[i] & \text{if } i = j, \\ \max \left\{ \begin{array}{l} c[i] + \min\Big(M[i+2,j], M[i+1,j-1]\Big), \\ c[j] + \min\Big(M[i+1,j-1], M[i,j-2]\Big) \end{array} \right\} & \text{if } i < j. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{CardGame}(c[0..n-1])$

Input: Array $c[0, n-1]$ representing the card values in sequence.
Output: Maximum points that can be made by player 1 $M[0..n-1]$

```
for i ← 0 to n − 1 do
  M[i, i] ← c[i]
  if i + 1 < n then M[i, i + 1] ← max(c[i], c[i + 1])
for k ← 2 to n − 1 do
  for i ← 0 to n − k do
    j ← i + k
    x ← c[i] + min(M[i + 2, j], M[i + 1, j − 1])
    y ← c[j] + min(M[i + 1, j − 1], M[i, j − 2])
    M[i, j] ← max(x, y)
return M[0, n − 1]
```

Step 6. Table

i	0	1	2	3	4	5
$c[i]$	12	13	4	2	1	10

$M[i,j]$	0	1	2	3	4	5
0	12	13	16	16	18	26
1		13	13	15	16	19
2			4	4	6	13
3				2	3	12
4					3	10
5						10

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Wildcard Pattern Matching HOME

Step 1. Problem

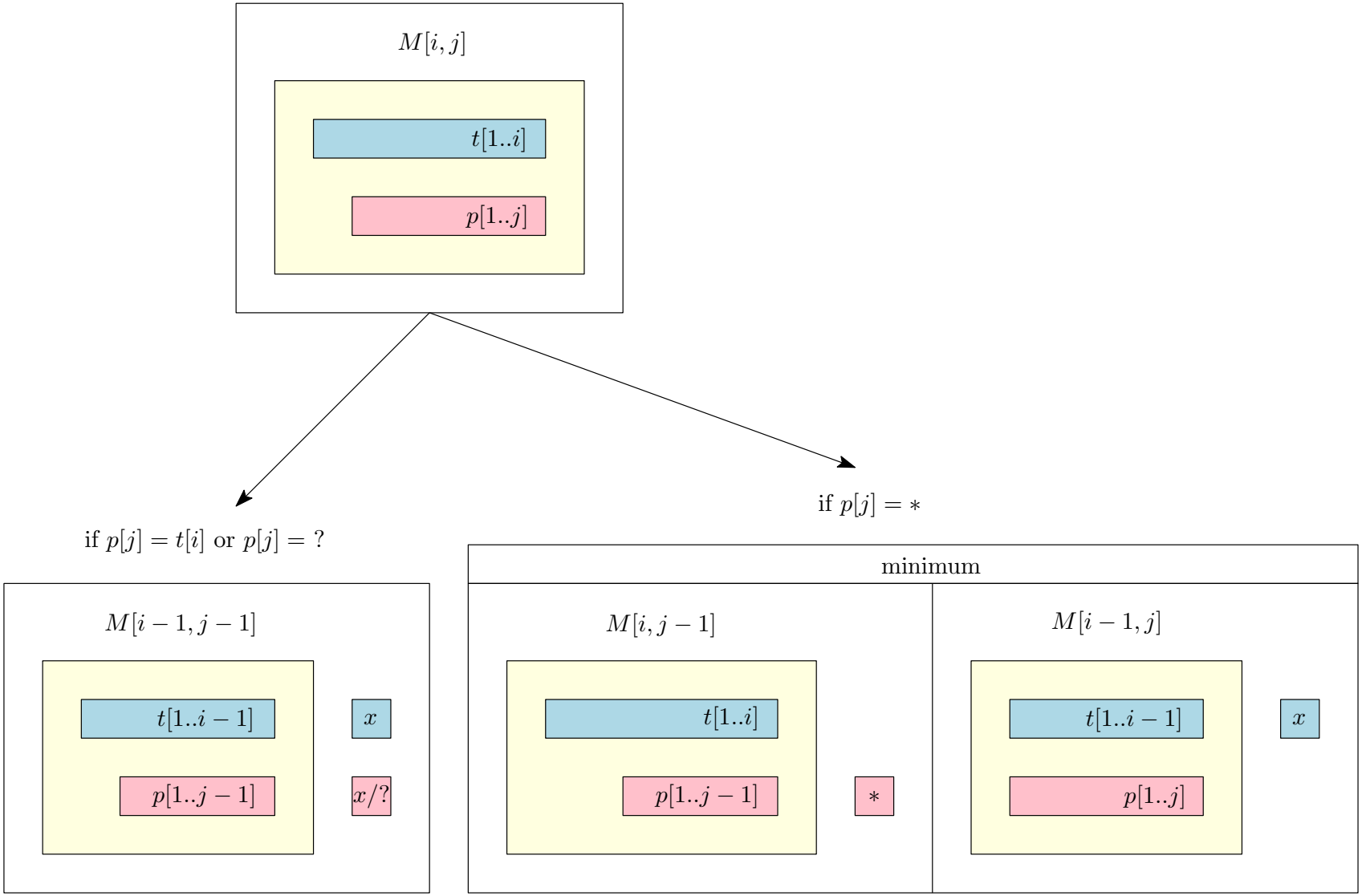
- [Link] Given a string of n characters $t[1..n]$ and a wildcard pattern of m characters $p[1..m]$, find if wildcard pattern matches the string.
- Matching should cover the entire string (not substring).
- The wildcard pattern can include $*$ (matches 0 or more characters) or $?$ (matches any single character). However, the string cannot include characters $*$ and $?$.
- For example, if the given string is $abbbabca$ and pattern is $?b*c?$, then the given pattern matches the string.

Step 2. Subproblem

$M[i, j] =$ Boolean value to represent if $t[1..i]$ matches $p[1..j]$.

Compute $M[n, m]$.

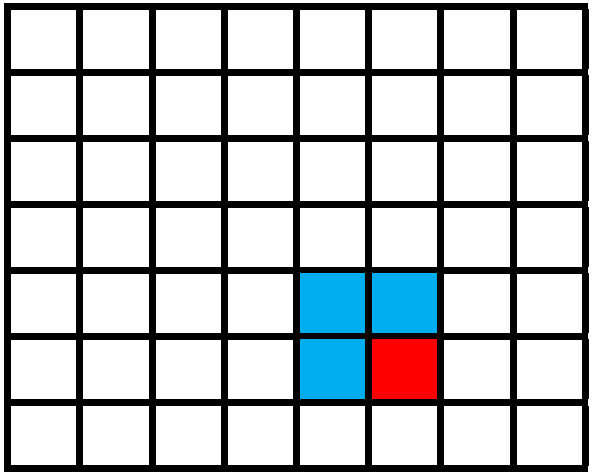
Step 3. Recurrence



Step 3. Recurrence

$$M[i,j] = \begin{cases} true & \text{if } i=0, j=0, \\ false & \text{if } j=0, i \in [1,n], \\ p[j] = * \text{ and } M[i,j-1] & \text{if } i=0, j \in [1,m], \\ \text{OR} \begin{cases} p[j] = t[i] \text{ and } M[i-1,j-1], \\ p[j] = ? \text{ and } M[i-1,j-1], \\ p[j] = * \text{ and } (M[i,j-1] \text{ or } M[i-1,j]) \end{cases} & \text{if } i \in [1,n], j \in [1,m]. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{WildcardPatternMatching}(t[1..n], p[1..m])$

Input: String $t[1..n]$ and pattern $p[1..m]$
Output: Boolean value $M[n, m]$ to represent if the string matches the pattern

```
M[0,0] ← true
for i ← 1 to n do M[i,0] ← false
for j ← 1 to m do
  if p[j] = * then M[0,j] ← M[0,j−1]
  else M[0,j] ← false
for i ← 1 to n do
  for j ← 1 to m do
    if p[j] = t[i] or p[j] =? then M[i,j] ← M[i−1,j−1]
    else if p[j] = * then M[i,j] ← M[i,j−1] or M[i−1,j]
    else M[i,j] ← false                                  ▷ p[j] ≠ t[i]
return M[n,m]
```

Step 6. Table

$M[i,j]$	0 ($p[0] = \varnothing$)	1 ($p[1] = ?$)	2 ($p[2] = b$)	3 ($p[3] = *$)	4 ($p[4] = c$)	5 ($p[5] = ?$)
0 ($t[0] = \varnothing$)	T	F	F	F	F	F
1 ($t[1] = a$)	F	T	T	F	F	F
2 ($t[2] = b$)	F	F	T	F	F	F
3 ($t[3] = b$)	F	F	T	F	F	F
4 ($t[4] = b$)	F	F	T	F	F	F
5 ($t[5] = a$)	F	F	T	F	F	F
6 ($t[6] = b$)	F	F	T	T	F	F
7 ($t[7] = c$)	F	F	T	F	T	F
8 ($t[8] = a$)	F	F	T	F	F	T

Step 7. Complexity

$$\text{Time} \in \Theta\left(nm\right), \text{Space} \in \Theta\left(nm\right)$$

Coin Toss HOME

Step 1. Problem

- [Link] Given n biased coins such that the probability of getting head from coin $i \in [1, n]$ is $p[i]$, compute the probability of getting exactly k heads when these n coins are tossed.

Step 2. Subproblem

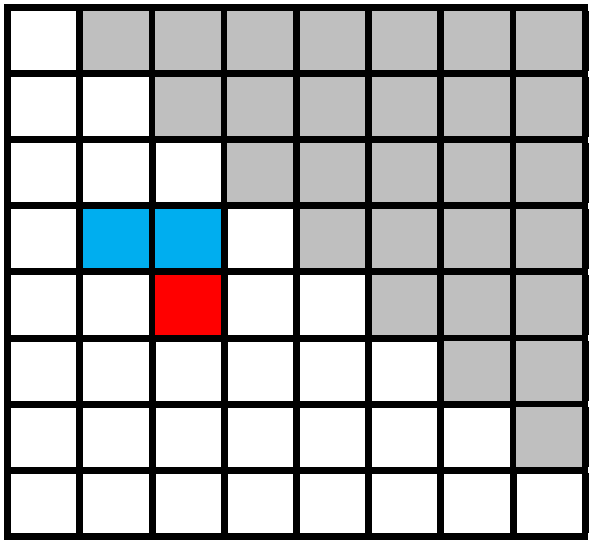
$P[i, j] =$ Probability of getting j heads when i coins are tossed.

Compute $P[n, k]$.

Step 3. Recurrence

$$P[i,j] = \left\{ \begin{array}{ll} 1 & \text{if } i = 0, j = 0, \\ 0 & \text{if } i \in [0,n], j \in [1,k], i < j, \\ (1 - p[i]) \cdot P[i-1,j] & \text{if } i \in [1,n], j = 0, \\ p[i] \cdot P[i-1,j-1] + (1 - p[i]) \cdot P[i-1,j] & \text{if } i \in [1,n], j \in [1,k], i \geq j. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{CoinToss}(p[1..n], k)$

Input: Array $p[1..n]$ such that $p[i]$ represents the probability of getting heads for biased coin i, and k is the number of heads desired.
Output: Probability of getting k heads with n biased coins.

```
P[0,0] ← 1
for j ← 1 to k do P[0,j] ← 0
for i ← 1 to n do
  P[i,0] ← (1 − p[i]) × P[i − 1,0]
  for j ← 1 to k do
    P[i,j] ← p[i] × P[i − 1,j − 1] + (1 − p[i]) × P[i − 1,j]
return P[n,k]
```

Step 6. Table

i	1	2	3	4	5	6
$p[i]$	0.2	0.4	0.5	0.7	0.6	0.1

$P[i,j]$	0	1	2	3	4	5	6
0	1						
1	0.8	0.2					
2	0.48	0.44	0.08				
3	0.24	0.46	0.26	0.4			
4	0.072	0.306	0.4	0.194	0.028		
5	0.0288	0.1656	0.3436	0.3176	0.1276	0.0168	
6	0.02592	0.15192	0.3258	0.3202	0.1466	0.02788	0.00168

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Dice Throw HOME

Step 1. Problem

- [Link] Given n dice, each with m faces, numbered from 1 to m, and a target sum s, find the number of ways in which the summation of values on all faces equals s.

Step 2. Subproblem

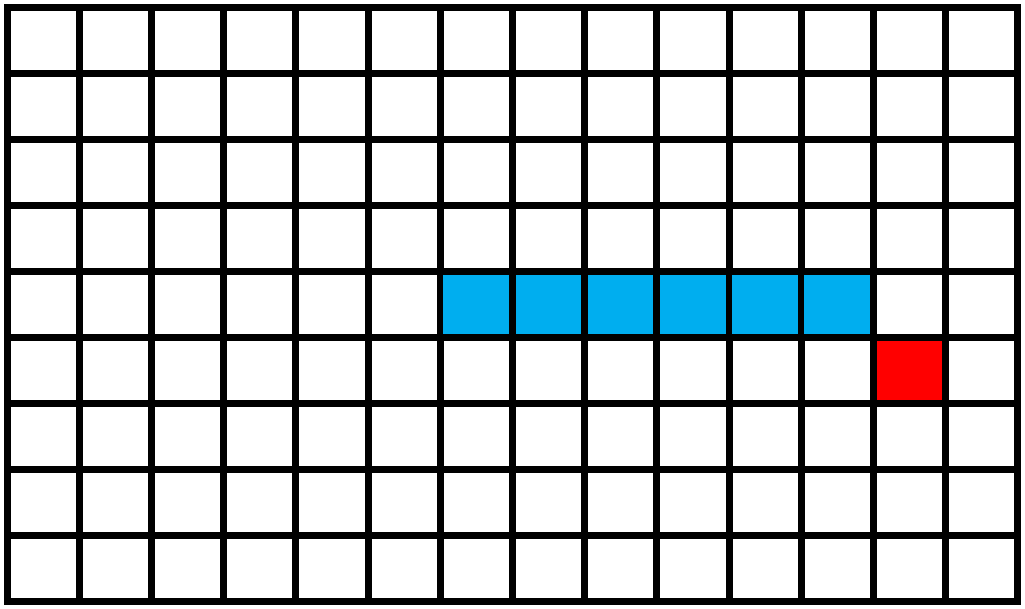
$C[i, j] =$ Number of ways in which the sum of values on the top faces of i dice equals j, assuming each die has m faces, numbered from 1 to m.

Compute $C[n, s]$.

Step 3. Recurrence

$$C[i,j] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \boxed{j \leq m} & \text{if } i = 1, j \in [1,s] \\ \left(\begin{array}{c} C[i-1,j-1] \times \boxed{j \geq 1} + \\ C[i-1,j-2] \times \boxed{j \geq 2} + \\ \cdots \\ C[i-1,j-m] \times \boxed{j \geq m} \end{array} \right) & \text{if } i \in [2,n], j \in [1,s]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{DiceThrow}(n, s)$

Input: n dice with m faces and sum s.
Output: Number of ways in which the sum of values on the top faces of n dice equals s, assuming each die has m faces, numbered from 1 to m.

```
if n × s = 0 then return 0
for j ← 1 to s do
  if j ≥ 1 and j ≤ m then C[1, j] ← 1
  else C[1, j] ← 0
for i ← 2 to n do
  for j ← 1 to s do
    C[i, j] ← 0
    for k ← 1 to m do
      if j − k ≥ 0 then
        C[i, j] ← C[i, j] + C[i − 1, j − k]
return C[n, s]
```

Step 6. Table

$$m = 6$$

$C[i,j]$	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	0	0	0	0
2		1	2	3	4	5	6	5	4	3
3			1	3	6	10	15	21	25	27
4				1	4	10	20	35	56	80
5					1	5	15	35	70	126
6						1	6	21	56	126
7							1	7	28	84
8								1	8	36
9									1	9
10										1

Step 7. Complexity

$$\text{Time} \in \Theta\left(nms\right), \text{Space} \in \Theta\left(ns\right)$$

Painter's Partition HOME

Step 1. Problem

- [Link] There are k painters who have to paint an array of n boards of lengths $a[1..n]$. Each painter can only paint a contiguous section (i.e., subarray) of these boards. Each painter takes a unit time to paint a unit of board. Find the minimum time required for these k painters to paint n boards.

Step 2. Subproblem

$T[i, j] =$ Minimum time required to paint i boards by j painters.

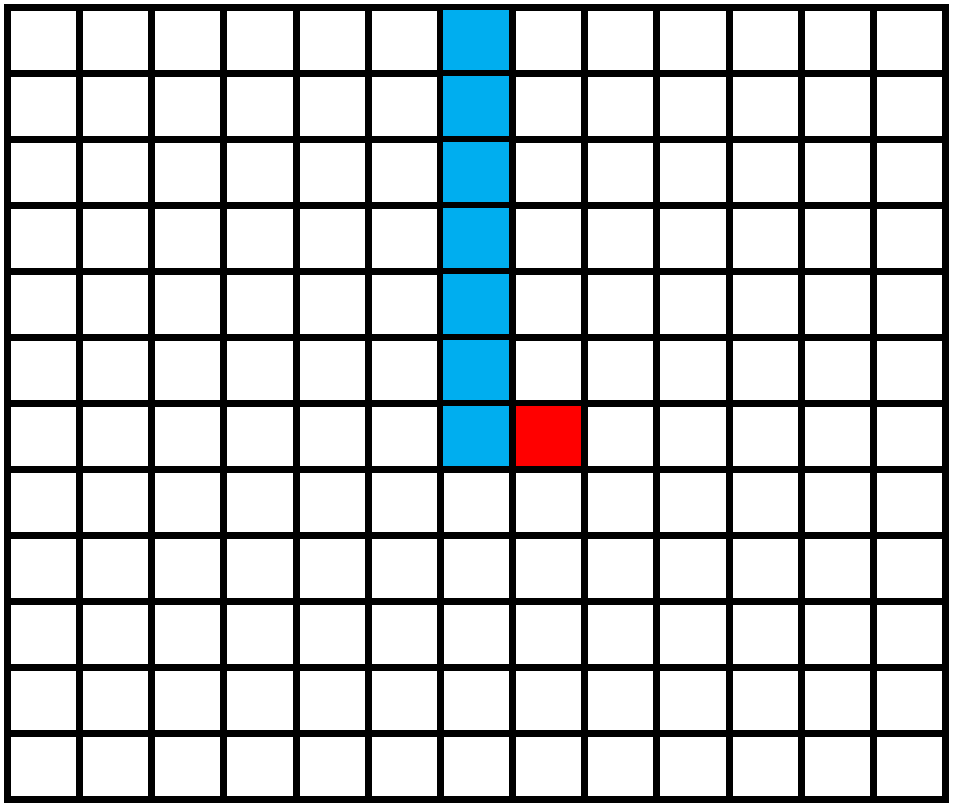
Compute $T[n, k]$.

Step 3. Recurrence

$$s[i,j] = a[i] + a[i+1] + a[i+2] + \cdots + a[j]$$

$$T[i,j] = \left\{ \begin{array}{ll} s[1,1] & \text{if } i = 1, j \in [1,k] \\ s[1,i] & \text{if } i \in [2,n], j = 1, \\ \min \left\{ \begin{array}{l} \max\left(T[1,j-1], s[2,i]\right), \\ \max\left(T[2,j-1], s[3,i]\right), \\ \cdots \\ \max\left(T[i-1,j-1], s[i,i]\right), \\ \max\left(T[i,j-1], s[i+1,i]\right) \end{array} \right\} & \text{if } i \in [2,n], j \in [2,k]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

PAINTER'SPARTITION$(a[1..n], k)$

Input: Lengths of paint boards $a[1..n]$ and number of painters k
Output: Minimum time required $T[n, k]$ to paint n boards by k painters

```
if k ≥ n then return max(a[1..n])                    ▷ special case
S[0] ← 0
for i ← 1 to n do s[i] ← s[i − 1] + a[i]             ▷ prefix sum
for j ← 1 to k do T[1, j] ← a[1]
for i ← 1 to n do T[i, 1] ← s[i]
for i ← 2 to n do
  for j ← 2 to k do
    result ← ∞
    for m ← 1 to i − 1 do
      result ← min(result, max(T[m, j − 1], s[i] − s[m]))
    T[i, j] ← result
return T[n, k]
```

Step 6. Table

i	1	2	3	4	5	6	7	8	9
$a[i]$	10	20	30	40	50	60	70	80	90
$s[i]$	10	30	60	100	150	210	280	360	450

$T[i,j]$	1	2	3	4	5	6	7	8	9
1	10	10	10	10	10	10	10	10	10
2	30	20	20	20	20	20	20	20	20
3	60	30	30	30	30	30	30	30	30
4	100	60	40	40	40	40	40	40	40
5	150	90	60	50	50	50	50	50	50
6	210	110	90	60	60	60	60	60	60
7	280	150	110	90	70	70	70	70	70
8	360	210	150	110	90	80	80	80	80
9	450	240	170	150	110	90	90	90	90

Step 7. Complexity

$$\text{Time} \in \mathcal{O}\left(n^2 k\right), \text{Space} \in \mathcal{O}\left(nk\right)$$

Filling Bookcase Shelves HOME

Step 1. Problem

- [Link] We are given a height-adjustable bookcase and a list of n books such that the height and width of the ith book are denoted by $h[i]$ and $w[i]$, respectively, where $i \in [1, n]$. We want to place these books onto bookcase shelves whose width is given as W. The order of the books we place must be the same as that of the given sequence of books. Find the minimum possible height of the height-adjustable bookcase.

Step 2. Subproblem

$H[i]$ = Minimum height of the height-adjustable bookcase for placing the first i books in order.

Compute $H[n]$.

Step 3. Recurrence

$$H[i] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, \\ \min\limits_{j \in [1,i] \text{ and } \boxed{\sum w[j..i] \leq W}} (H[j-1] + \max h[j..i]) & \text{if } i \in [1,n]. \end{array} \right\}$$

Step 4. Dependency

Step 5. Algorithm

$\textsc{FillingBookcaseShelves}(h[1..n], w[1..n], W)$

Input: Arrays $h[1..n]$ and $w[1..n]$ denoting height and width of each book, respectively, and bookcase shelf width W

Output: Minimum possible height of bookcase $H[n]$ to the place the n books in order

```
H[0] ← 0
for i ← 1 to n do
  H[i] ← ∞; currh ← 0; currw ← 0
  for j ← i to 1 do
    currw ← currw + w[j]
    if currw > W then break
    currh ← max(h[j], currh)
    H[i] ← min(H[i], currh + H[j − 1])
return H[n]
```

Step 6. Table

Bookcase shelf width $W = 4$

i	1	2	3	4	5	6	7
$w[i]$	1	2	2	1	1	1	1
$h[i]$	1	3	3	1	1	1	2
$H[i]$	1	2	4	5	5	5	6

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n\right)$$

Gold Mine HOME

Step 1. Problem

- [Link] Given an $m \times n$ 2-D matrix, where each cell in the matrix contains a positive integer denoting the tonnes of gold available at the current spot. A miner begins at any row within the 0th column and is allowed to only move right, diagonally up right or diagonally down right.
- Compute the maximum ton of gold that can be collected by a miner moving through the matrix till the last column.

Step 2. Subproblem

$$g[i,j] = \text{Gold at } [i,j]$$

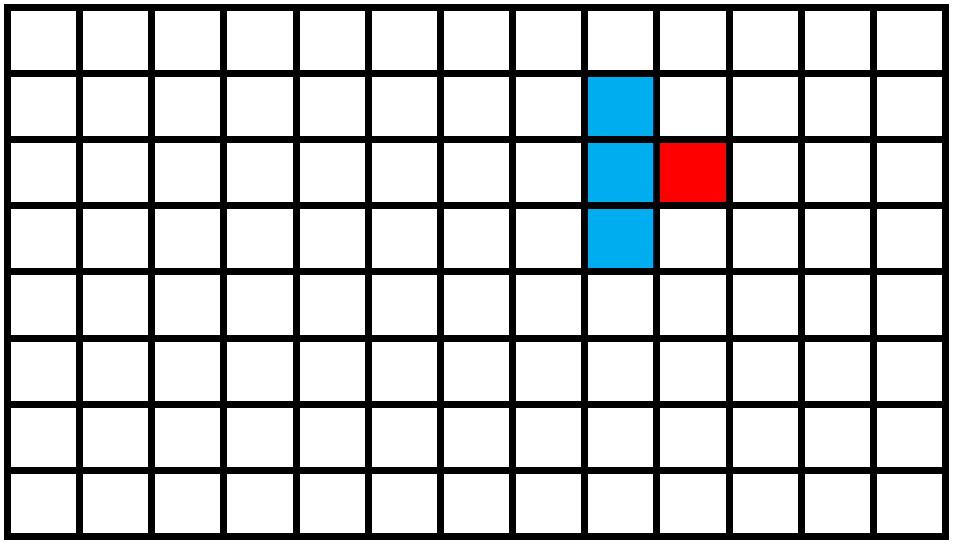
$$M[i,j] = \text{Maximum gold collected at } [i,j] \text{ using right, right up or right down directions.}$$

Compute $\max(M[0..m-1, n-1])$

Step 3. Recurrence

$$M[i,j] = \left\{ \begin{array}{ll} g[i,j] & \text{if } j = 0, \\ g[i,j] + & \\ \max \left\{ \begin{array}{l} M[i-1,j-1] \times \boxed{i \geq 1} \\ M[i,j-1] \\ M[i+1,j-1] \times \boxed{i \leq m-2} \end{array} \right\} & \text{if } j \in [1, n-1]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{GoldMine}(g[0..m-1, 0..n-1])$

Input: 2-D matrix $g[0..m-1, 0..n-1]$
Output: Maximum amount of gold that can be collected by traversing through the matrix.

```
M ← g
for j ← 1 to n − 1 do
  for i ← 0 to m − 1 do
    lu ← (i ≥ 1) ? M[i − 1, j − 1] : 0          ▷ left up
    l ← M[i, j − 1]                              ▷ left
    ld ← (i ≤ m − 2) ? M[i + 1, j − 1] : 0       ▷ left down
    M[i, j] ← M[i, j] + max(lu, l, ld)
return max(M[0..m − 1, n − 1])
```

Step 6. Table

$g[i,j]$	0	1	2	3
0	1	3	1	5
1	2	2	4	1
2	5	0	2	3
3	0	6	1	2

$M[i,j]$	0	1	2	3
0	1	5	8	16
1	2	7	11	14
2	5	5	13	16
3	0	11	12	15

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(mn\right)$$

Word Break HOME

Step 1. Problem

- [Link] Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words D.

Step 2. Subproblem

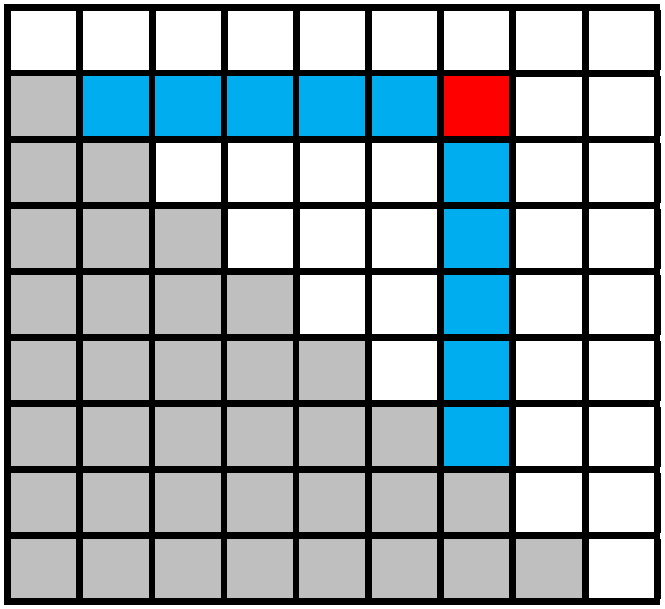
$B[i, j] =$ Boolean value to represent if the input substring $s[i..j]$ can be segmented into space-separated sequence of dictionary words D.

Compute $B[0, n-1]$.

Step 3. Recurrence

$$B[i,j] = \begin{cases} \boxed{s[i] \in D} & \text{if } i = j, \\ \text{OR} \begin{cases} \boxed{s[i..j] \in D}, \\ (B[i,i] \text{ and } B[i+1,j]), \\ (B[i,i+1] \text{ and } B[i+2,j]), \\ \dots \\ (B[i,j-1] \text{ and } B[j,j]), \end{cases} & \text{if } i < j. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

WordBreak($s[0..n-1], D$)

Input: An input string $s[0..n-1]$ and a dictionary of words D
Output: Boolean value $B[0, n-1]$ to represent if the input string S can be segmented into space-separated sequence of dictionary words

```
for i ← 0 to n − 1 do
  if s[i] ∈ D then B[i, i] ← true
  else if s[i] ∉ D then B[i, i] ← false
for ℓ ← 2 to n do
  for i ← 0 to n − ℓ do
    j ← i + ℓ − 1
    if s[i..j] ∈ D then
      B[i, j] ← true
    else if s[i..j] ∉ D then
      B[i, j] ← false
      for k ← i to j − 1 do
        if B[i, k] = true and B[k + 1, j] = true then
          B[i, j] ← true; break
return B[0, n − 1]
```

Step 6. Table

$$D = \{I, a, am, ace\}$$
$$s = Iamace$$

$B[i,j]$	0 ($s[0] = I$)	1 ($s[1] = a$)	2 ($s[2] = m$)	3 ($s[3] = a$)	4 ($s[4] = c$)	5 ($s[5] = e$)
0 ($s[0] = I$)	✓	✓	✓	✓	✗	✓
1 ($s[1] = a$)		✓	✓	✓	✗	✓
2 ($s[2] = m$)			✗	✗	✗	✗
3 ($s[3] = a$)				✓	✗	✓
4 ($s[4] = c$)					✗	✗
5 ($s[5] = e$)						✗

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^3\right), \text{Space} \in \Theta\left(n^2\right)$$

Step 2. Subproblem

$B[i]$ = Boolean value to represent if the input substring $s[1..i]$ can be segmented into space-separated sequence of dictionary words D.

Compute $B[n]$.

Step 3. Recurrence

$$B[i] = \begin{cases} true & \text{if } i = 0, \\ \text{OR} \begin{cases} (B[0] \text{ and } \boxed{s[1..i] \in D}), \\ (B[1] \text{ and } \boxed{s[2..i] \in D}), \\ \dots \\ (B[i-1] \text{ and } \boxed{s[i..i] \in D}), \end{cases} & \text{if } i \in [1, n]. \end{cases}$$

Step 4. Dependency

Step 5. Algorithm

$\textsc{WordBreak}(s[1..n], D)$

Input: An input string $s[1..n]$ and a dictionary of words D

Output: Boolean value $B[n]$ to represent if the input string s can be segmented into space-separated sequence of dictionary words

```
B[0] ← true
for i ← 1 to n do
  for j ← 1 to i do
    if B[j] = true and s[j..i] ∈ D then
      B[i] = true; break
return B[n]
```

Step 6. Table

$$D = \{I, a, am, ace\}$$
$$s[1..6] = Iamace$$

i	0	1	2	3	4	5	6
$B[i]$	✓	✓	✓	✓	✓	✗	✓

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n\right)$$

Combinations HOME

Step 1. Problem

- [Link] Find the number of ways to construct n-sized non-decreasing arrays from numbers $\{1, 2, \ldots, v\}$ considering each element can be used at most k times.
- Suppose $n = 3$, $v = 3$, $k = 2$, then output is 7 because we have the following seven 3-sized arrays: $[1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3], [1, 3, 3], [2, 2, 3], [2, 3, 3]$.

Step 2. Subproblem

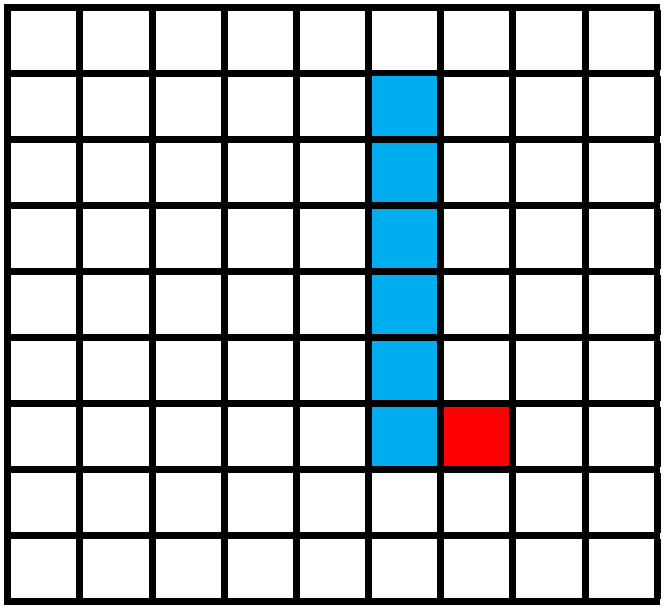
$C[i, j] =$ Number of non-decreasing sequences of size i using numbers from 1 to j, considering each element can be used at most k times.

Compute $C[n, v]$.

Step 3. Recurrence

$$C[i,j] = \left\{ \begin{array}{ll} 1 & \text{if } i = 0, \\ 0 & \text{if } i \in [1,n], j = 0, \\ \left\{ \begin{array}{l} (C[i,j-1]) + \\ (C[i-1,j-1]) + \\ (C[i-2,j-1]) \times \boxed{i \geq 2} + \\ \vdots \\ (C[i-k,j-1]) \times \boxed{i \geq k} \end{array} \right\} & \text{if } i \in [1,n], j \in [1,v]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{Combinations}(n, v, k)$

Input: Array size n, elements $\{1, 2, \ldots, v\}$, number of occurrences of each element is at most k.

Output: Count $C[n, v]$ the number of non-decreasing n-sized arrays that can be constructed from the elements $\{1, 2, \ldots, v\}$ considering at most k occurrences of each element.

Notation: $S[i, j] = C[0, j] + C[1, j] + \cdots + C[i, j]$.

```
for j ← 0 to v do C[0, j] ← 1; S[0, j] ← 1
for i ← 1 to n do C[i, 0] ← 0; S[i, 0] ← 1
for j ← 1 to v do
  for i ← 1 to n do
    C[i, j] ← S[i, j − 1]
    if i > k then C[i, j] ← C[i, j] − S[i − k − 1, j − 1]
    S[i, j] ← S[i − 1, j] + C[i, j]
return C[n, v]
```

Step 6. Table

$$n = 3,\ v = 3,\ k = 2$$

$C[i,j]$	0	1	2	3
0	1	1	1	1
1	0	1	2	3
2	0	1	3	6
3	0	0	2	7

$S[i,j]$	0	1	2	3
0	1	1	1	1
1	1	2	3	4
2	1	3	6	10
3	1	3	8	17

Step 7. Complexity

$$\text{Time} \in \Theta\left(nv\right), \text{Space} \in \Theta\left(nv\right)$$

Ugly Number HOME

Step 1. Problem

- [Link] An ugly number is a number whose only prime factors are 2, 3, or 5. The first few ugly numbers are as follows:

i	0	1	2	3	4	5	6	7	8	9	10
$U[i]$	1	2	3	4	5	6	8	9	10	12	15

Find the nth ugly number.

Step 2. Subproblem

$U[i] = i$th ugly number.

Compute $U[n]$.

Step 3. Recurrence

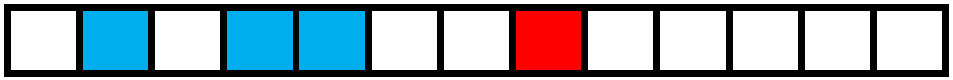
$$U[i] = \left\{ \begin{array}{ll} 1 & \text{if } i = 0, \\ \min\{U[i2[i-1]] \times 2, U[i3[i-1]] \times 3, U[i5[i-1]] \times 5\} & \text{if } i \in [1, n]. \end{array} \right\}$$

$$i2[i] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, \\ i2[i-1] + \boxed{U[i] = U[i2[i-1]] \times 2} & \text{if } i \in [1, n]. \end{array} \right\}$$

$$i3[i] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, \\ i3[i-1] + \boxed{U[i] = U[i3[i-1]] \times 3} & \text{if } i \in [1, n]. \end{array} \right\}$$

$$i5[i] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, \\ i5[i-1] + \boxed{U[i] = U[i5[i-1]] \times 5} & \text{if } i \in [1, n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

UglyNumber(n)

Input: Number n
Output: nth ugly number $U[n]$

```
U[0] ← 1                                   ▷ Ugly numbers start from index 0
i2 ← 0;  i3 ← 0;  i5 ← 0
for i ← 1 to n do
    U[i] ← min(U[i2] × 2,  U[i3] × 3,  U[i5] × 5)
    if U[i] = U[i2] × 2 then i2 ← i2 + 1
    if U[i] = U[i3] × 3 then i3 ← i3 + 1
    if U[i] = U[i5] × 5 then i5 ← i5 + 1
return U[n]
```

Step 6. Table

$$n = 10$$

i	0	1	2	3	4	5	6	7	8	9	10
$U[i]$	1	2	3	4	5	6	8	9	10	12	15

Step 7. Complexity

$$\text{Time} \in \Theta\left(n\right), \text{Space} \in \Theta\left(n\right)$$

Parenthesizations HOME

Step 1. Problem

- [Link] Count the number of possible valid parenthesizations given n open and n close parentheses.
- When $n = 3$, $count = 5$.

$$\text{Parenthesizations} = \left\{ \begin{array}{ccccccc} (& (& (&) &) &) \\ (&) & (& (&) &) \\ (& (&) &) & (&) \\ (& (&) & (&) &) \\ (&) & (&) & (&) \end{array} \right\}$$

Step 2. Subproblem

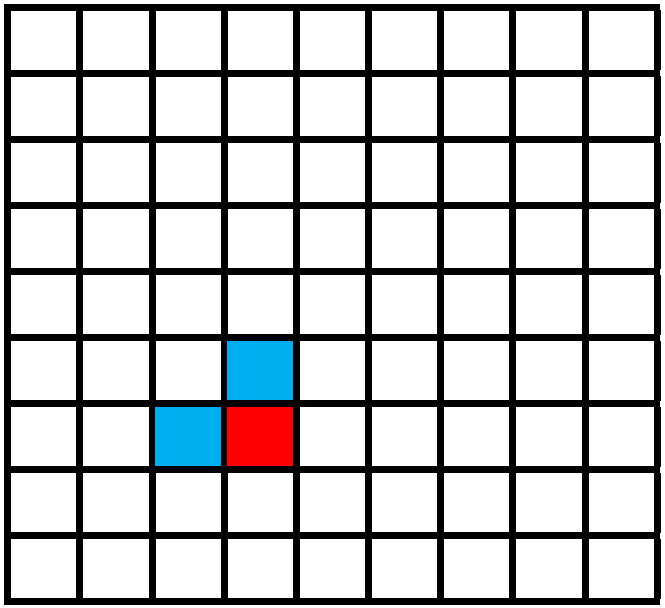
$$C[i,j] = \#\text{Parenthesizations (valid) with } i \text{ open and } j \text{ close parentheses.}$$

Compute $C[n,n]$.

Step 3. Recurrence

$$C[i,j] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, j \in [1,n] \\ \boxed{i \geq 1} & \text{if } i \in [1,n], j = 0, \\ \boxed{i \geq j} \times C[i-1,j] + \boxed{i > j} \times C[i,j-1] & \text{if } i,j \in [1,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

PARENTHESIZATIONS(n)

Input: Number of open and close parentheses n

Output: Number of valid unique parenthesizations using n open and n close parentheses

```
for j ← 0 to n do C[0, j] ← 0
for i ← 1 to n do C[i, 0] ← 1
for i ← 1 to n do
  for j ← 1 to n do
    C[i, j] ← 0
    if i ≥ j then C[i, j] ← C[i, j] + C[i − 1, j]
    if i > j then C[i, j] ← C[i, j] + C[i, j − 1]
return C[n, n]
```

Step 6. Table

$C[i,j]$	0	1	2	3	4	5	6	7	8
0	0								
1	1	1							
2	1	2	2						
3	1	3	5	5					
4	1	4	9	14	14				
5	1	5	14	28	42	42			
6	1	6	20	48	90	132	132		
7	1	7	27	75	165	297	429	429	
8	1	8	35	110	275	572	1001	1430	1430

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Step 1. Problem

- [Link] Count the number of valid parenthesizations given n open and close parentheses.
- When $n = 3$, $count = 5$.

$$\text{Parenthesizations} = \left\{ \begin{array}{ccccccc} (& (& (&) &) &) \\ (&) & (& (&) &) \\ (& (&) &) & (&) \\ (& (&) & (&) &) \\ (&) & (&) & (&) \end{array} \right\}$$

Step 2. Subproblem

$C[i] = \#$Parenthesizations (valid) with i open and close parentheses.

Compute $C[n]$.

Step 3. Recurrence

$$C[i] = \left\{ \begin{array}{ll} 1 & \text{if } i = 0, \\ \left\{ \begin{array}{c} C[0] \times C[i-1] + \\ C[1] \times C[i-2] + \\ \cdots \\ C[i-1] \times C[0] \end{array} \right\} & \text{if } i \in [1, n]. \end{array} \right\}$$

Step 4. Dependency

Step 5. Algorithm

PARENTHESIZATIONS(n)

Input: Number of open and close parentheses n

Output: Number of valid unique parenthesizations using n open and n close parentheses

```
C[0] ← 1
for i ← 1 to n do
  C[i] ← 0
  for j ← 0 to i − 1 do
    C[i] ← C[i] + C[j] × C[i − j − 1]
return C[n]
```

Step 6. Table

i	0	1	2	3	4	5	6	7
$C[i]$	1	1	2	5	14	42	132	429

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n\right)$$

Derivations using Ambiguous Grammar HOME

Step 1. Problem

- [Link] Count the number of different leftmost derivations (or LMDs) for the string $a + a + \cdots + a$, where the operand a is repeated n times using the below ambiguous grammar. In the grammar, E stands for arithmetic expression.

$$E \to E + E \mid a$$

Step 2. Subproblem

$C[i] =$ Number of different derivations for string with i operands.

Compute $C[n]$.

Step 3. Recurrence

$$C[i] = \left\{ \begin{array}{ll} 1 & \text{if } i = 1, \\ \left(\begin{array}{c} C[1] \times C[i-1] + \\ C[2] \times C[i-2] + \\ \cdots \\ C[i-1] \times C[1] \end{array} \right) & \text{if } i \in [2, n]. \end{array} \right\}$$

Step 4. Dependency

Step 5. Algorithm

$\textsc{CountLeftmostDerivations}(n)$

Input: The number of operands n in the string $a + a + \cdots + a$.
Output: Number of leftmost derivations of string $a + a + \cdots + a$ with n operands using ambiguous grammar $E \to E + E \mid a$.

```
C[1] ← 1
for i ← 2 to n do
  C[i] ← 0
  for j ← 1 to i − 1 do
    C[i] ← C[i] + C[j] × C[i − j]
return C[n]
```

Step 6. Table

i	1	2	3	4	5	6	7	8
$C[i]$	1	1	2	5	14	42	132	429

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n\right)$$

Optimal Binary Search Tree HOME

Step 1. Problem

- [Link] Given a sorted search keys array of n real numbers $k[1..n]$ and a successful key search probability array with probabilities $p[1..n]$, unsuccessful key search probability array with probabilities $q[0..n]$ respectively, where $p[i]$ is the probability that key search will take place for key $k[i]$, $q[i]$ where $i \in [1, n-1]$ is the probability that key search will take place for a key between $k[i]$ and $k[i+1]$. Note that sum$(p[1..n])$ + sum$(q[0..n]) = 1$. Also, $q[0]$ is the probability that key search will take place for a key less than $k[0]$, $q[n]$ is the probability that key search will take place for a key greater than $k[n]$. The task is to construct a binary search tree from the keys in k such that the keys search cost is minimized, also note that the level of the root is 1. Search cost of a binary search tree is defined as below.

$$\text{Search Cost} = \sum_{i=1}^{n}((depth(k[i])+1)\times p[i]) + \sum_{i=0}^{n}((depth(\varnothing)+1)\times q[i]).$$

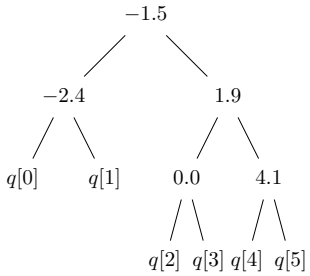
Step 1. Problem

- For example if we have the following,

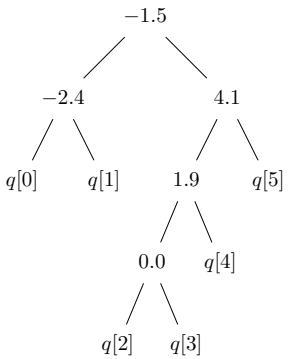
Array	Contents
Keys $k[1..5]$	$[-2.4, -1.5, 0.0, 1.9, 4.1]$
Successful key search prob. $p[1..5]$	$[0.15, 0.10, 0.05, 0.10, 0.20]$
Unsuccessful key search prob. $q[0..5]$	$[0.05, 0.10, 0.05, 0.05, 0.05, 0.10]$

then the minimum search cost for the optimal BST is 2.75.

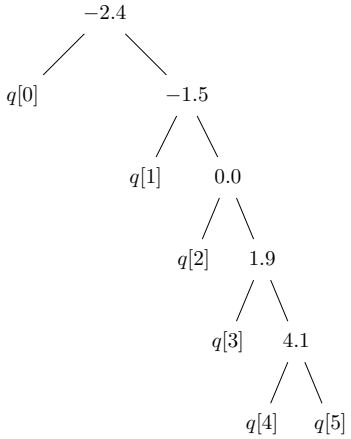
- Below are the few possible BSTs with search costs.



Search Cost is 2.80



Search Cost is 2.75



Search Cost is 3.65

Step 2. Subproblem

$C[i, j] =$ Optimal cost of searching a binary search tree containing keys $k[i..j]$ with respective successful key search probabilities $p[i..j]$, unsuccessful key search probabilities $q[i-1..j]$.

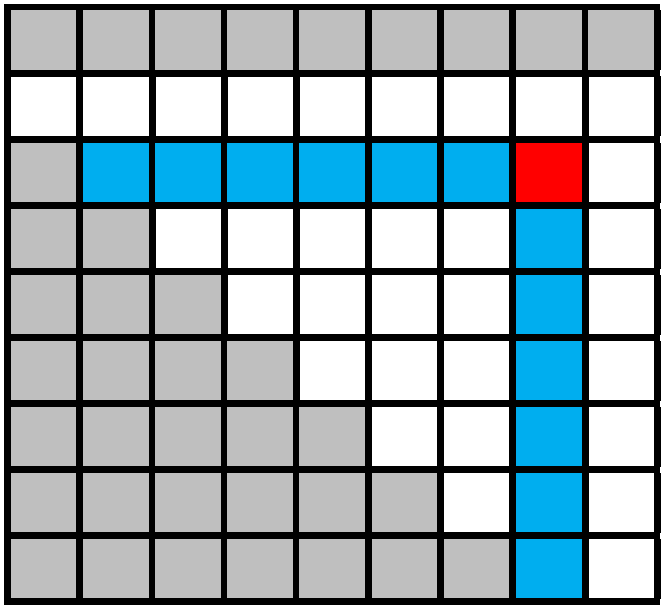
Compute $C[1, n]$.

Step 3. Recurrence

$$i \in [1, n+1],\ j \in [0, n]$$

$$C[i,j] = \left\{ \begin{array}{ll} q[i-1] & \text{if } i = j+1, \\ \begin{Bmatrix} \mathsf{sum}(p[i..j]) + \\ \mathsf{sum}(q[i-1..j]) \end{Bmatrix} + \min \begin{Bmatrix} C[i,i-1] + C[i+1,j], \\ C[i,i] + C[i+2,j], \\ \vdots \\ C[i,j-1] + C[j+1,j] \end{Bmatrix} & \text{if } i \leq j. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{OptimalBST}(k[1..n], p[1..n], q[0..n])$

Input: Sorted search keys array $k[1..n]$, successful key search probability array $p[1..n]$, unsuccessful key search probability array $q[0..n]$.
Output: Minimum search cost $C[1, n]$ obtained through optimal way of arranging keys in a BST.
Notation: $w[i, j] = p[i] + p[i+1] + \ldots + p[j] + q[i-1] + q[i] + \ldots + q[j]$.

```
for i ← 1 to n + 1 do
  C[i, i − 1] ← q[i − 1]; w[i, i − 1] ← q[i − 1]
for ℓ ← 1 to n do
  for i ← 1 to n − ℓ + 1 do
    j ← i + ℓ − 1; C[i, j] ← ∞; w[i, j] ← w[i, j − 1] + p[j] + q[j]
    for index ← i to j do
    {
      minimum ← C[i, index − 1] + C[index + 1, j] + w[i, j]
      C[i, j] ← min(C[i, j], minimum)
    }
return C[1, n]
```

Step 6. Table

i	0	1	2	3	4	5
$p[i]$	-	0.15	0.10	0.05	0.10	0.20
$q[i]$	0.05	0.10	0.05	0.05	0.05	0.10

$C[i,j]$	0 ($k[0] = \varnothing$)	1 ($k[1] = -2.4$)	2 ($k[2] = -1.5$)	3 ($k[3] = 0.0$)	4 ($k[4] = 1.9$)	5 ($k[5] = 4.1$)
0 ($k[0] = \varnothing$)						
1 ($k[1] = -2.4$)	0.05	0.45	0.90	1.25	1.75	2.75
2 ($k[2] = -1.5$)		0.10	0.40	0.70	1.20	2.00
3 ($k[3] = 0.0$)			0.05	0.25	0.60	1.30
4 ($k[4] = 1.9$)				0.05	0.30	0.90
5 ($k[5] = 4.1$)					0.05	0.50
6 ($k[6] = \varnothing$)						0.10

$w[i,j]$	0	1	2	3	4	5
0						
1	0.05	0.30	0.45	0.55	0.70	1.00
2		0.10	0.25	0.35	0.50	0.80
3			0.05	0.15	0.30	0.60
4				0.05	0.20	0.50
5					0.05	0.35
6						0.10

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^3\right), \text{Space} \in \Theta\left(n^2\right)$$

Step 2. Subproblem

$C[i, j] =$ Optimal cost of searching a binary search tree containing keys $k[i..j]$ with respective successful key search probabilities $p[i..j]$, unsuccessful key search probabilities $q[i-1..j]$.

Compute $C[1, n]$.

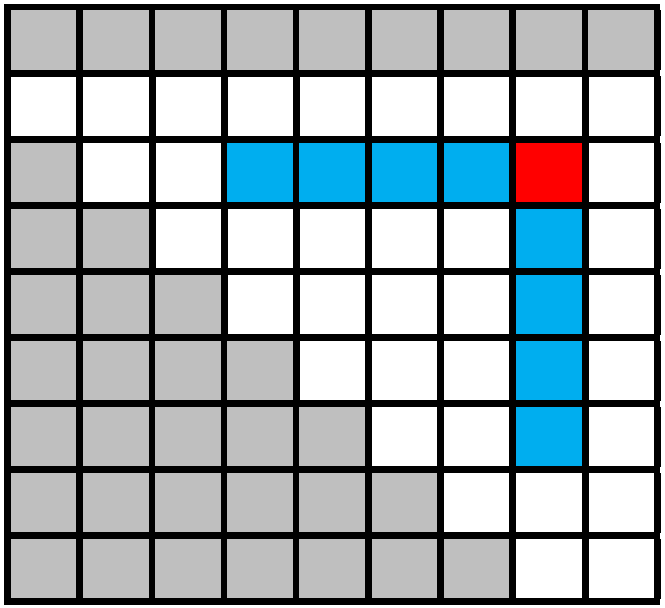
Step 3. Recurrence

$$i \in [1, n+1],\ j \in [0, n]$$

$$C[i,j] = \left\{ \begin{array}{ll} q[i-1] & \text{if } i = j+1, \\ \mathsf{sum}(p[i..j]) + \mathsf{sum}(q[i-1..j]) \times 2 & \text{if } i = j, \\ \left\{ \begin{array}{l} \mathsf{sum}(p[i..j]) + \\ \mathsf{sum}(q[i-1..j]) \end{array} \right\} + \min_{t \in [r[i,j-1], r[i+1,j]]} (C[i,t-1], C[t+1,j]) & \text{if } i < j. \end{array} \right\}$$

where $r[i,j]$ is the root index of the BST for which the minimum optimal search cost is $C[i,j]$

Step 4. Dependency



Step 5. Algorithm

$\textsc{OptimalBST}(k[1..n], p[1..n], q[0..n])$

Input: Sorted search keys array $k[1..n]$, successful key search probability array $p[1..n]$, unsuccessful key search probability array $q[0..n]$.
Output: Minimum search cost $C[1, n]$ obtained through optimal way of arranging keys in a BST.
Notation: $w[i, j] = p[i] + p[i+1] + \ldots + p[j] + q[i-1] + q[i] + \ldots + q[j]$.
Notation: $r[i, j]$ contains the index of the root at which $C[i, j]$ achieves min.

```
for i ← 1 to n + 1 do
  C[i, i − 1] ← q[i − 1]; w[i, i − 1] ← q[i − 1]
  if i ≤ n then r[i, i] ← i
for ℓ ← 1 to n do
  for i ← 1 to n − ℓ + 1 do
    j ← i + ℓ − 1; C[i, j] ← ∞; w[i, j] ← w[i, j − 1] + p[j] + q[j]
    if ℓ = 1 then C[i, j] ← w[i, j] + q[j − 1] + q[j]; continue
    for index ← r[i, j − 1] to r[i + 1, j] do
    {
      minimum ← C[i, index − 1] + C[index + 1, j] + w[i, j]
      if minimum < C[i, j] then C[i, j] ← minimum; r[i, j] ← index
    }
return C[1, n]
```

Step 6. Table

i	0	1	2	3	4	5
$p[i]$	-	0.15	0.10	0.05	0.10	0.20
$q[i]$	0.05	0.10	0.05	0.05	0.05	0.10

$C[i,j]$	0 ($k[0] = \varnothing$)	1 ($k[1] = -2.4$)	2 ($k[2] = -1.5$)	3 ($k[3] = 0.0$)	4 ($k[4] = 1.9$)	5 ($k[5] = 4.1$)
0 ($k[0] = \varnothing$)						
1 ($k[1] = -2.4$)	0.05	0.45	0.90	1.25	1.75	2.75
2 ($k[2] = -1.5$)		0.10	0.40	0.70	1.20	2.00
3 ($k[3] = 0.0$)			0.05	0.25	0.60	1.30
4 ($k[4] = 1.9$)				0.05	0.30	0.90
5 ($k[5] = 4.1$)					0.05	0.50
6 ($k[6] = \varnothing$)						0.10

$w[i,j]$	0	1	2	3	4	5
0						
1	0.05	0.30	0.45	0.55	0.70	1.00
2		0.10	0.25	0.35	0.50	0.80
3			0.05	0.15	0.30	0.60
4				0.05	0.20	0.50
5					0.05	0.35
6						0.10

$r[i,j]$	1	2	3	4	5
1	1	1	2	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Snakes and Ladders HOME

Step 1. Problem

- [Link] Given a board $b[1..n]$, reachable $r[1..n]$, find the minimum number of dice throws required to reach the destination n (≥ 6) from origin on a snakes and ladders board. A cell cannot simultaneously contain two or more elements from the set $\{$snake head, snake tail, ladder top, ladder bottom$\}$.

$$b[i] = \left\{ \begin{array}{ll} i & \text{if cell } i \text{ neither contains snake tail nor ladder top,} \\ k & \text{if there is a ladder from cell } i \text{ to cell } k \ (k < i), \\ k & \text{if the cell has snake head at cell } i \text{ leading to cell } k \ (> i). \end{array} \right\}$$

$$r[i] = \left\{ \begin{array}{ll} i & \text{if cell } i \text{ neither contains snake tail nor ladder top,} \\ k & \text{if there is a ladder from cell } k \text{ to cell } i \ (k < i), \\ k & \text{if the cell has snake head at cell } k \text{ leading to cell } i \ (< k). \end{array} \right\}$$

Step 2. Subproblem

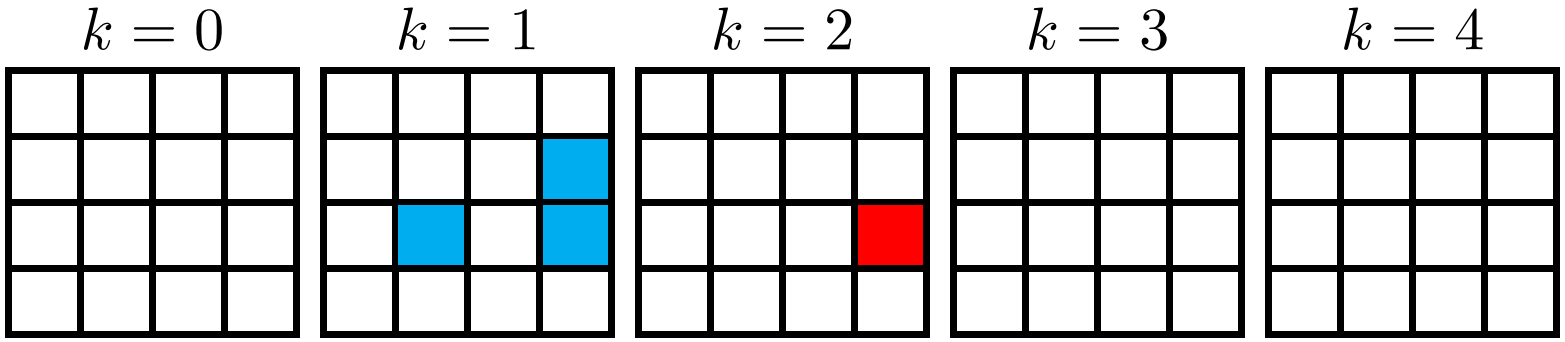
$M[k, i, j] =$ Minimum number of dice throws required to reach cell i from cell j after considering that any of $1..k$ cells exist in the path between visiting i and j.

Compute $M[n, 1, n]$.

Step 3. Recurrence

$$M[k,i,j] = \left\{ \begin{array}{ll} 0 & \text{if } k = 0 \text{ and } \{i = j \text{ or } b[i] = j\}, \\ 1 & \text{if } k = 0 \text{ and } \{j - i \in [1,6] \text{ and } b[i] = i\}, \\ \infty & \text{if } k = 0, \\ \min(M[k-1,i,j], M[k-1,i,k] + M[k-1,k,j]) & \text{if } k \in [1,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

SnakeAndLadders($b[1..n]$)

Input: Board $b[1..n]$ with snakes and ladders.
Output: Minimum number of dice throws needed to reach n.

```
for i ← 1 to n do
  for j ← 1 to n do
    if b[i] = j or i = j do M[i, j] ← 0
    else if b[i] = i and j − i ≥ 1 and j − i ≤ 6 do M[i, j] ← 1
    else M[i, j] ← ∞
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      M[i, j] ← min(M[i, j], M[i, k] + M[k, j])
return M[1, n]
```

Step 6. Table

i	1	2	3	4	5	6	7	8	9
$b[i]$	1	2	3	9	7	6	7	8	9

$M[9, i, j]$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	∞	1	1	1	1	1	1	1	1	1
2	∞	1	1	1	1	1	1	1	1	1
3	∞	1	1	1	1	1	1	1	1	1
4	∞	1	1	1	1	1	1	1	1	1
5	∞	1	1	1	1	1	1	1	1	1
6	∞	1	1	1	1	1	1	1	1	1
7	∞	∞	1	1	1	1	1	1	1	1
8	∞	∞	2	2	2	2	2	2	2	2
9	∞	∞	1	1	1	1	1	1	1	1

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^3\right), \text{Space} \in \Theta\left(n^2\right)$$

Step 2. Subproblem

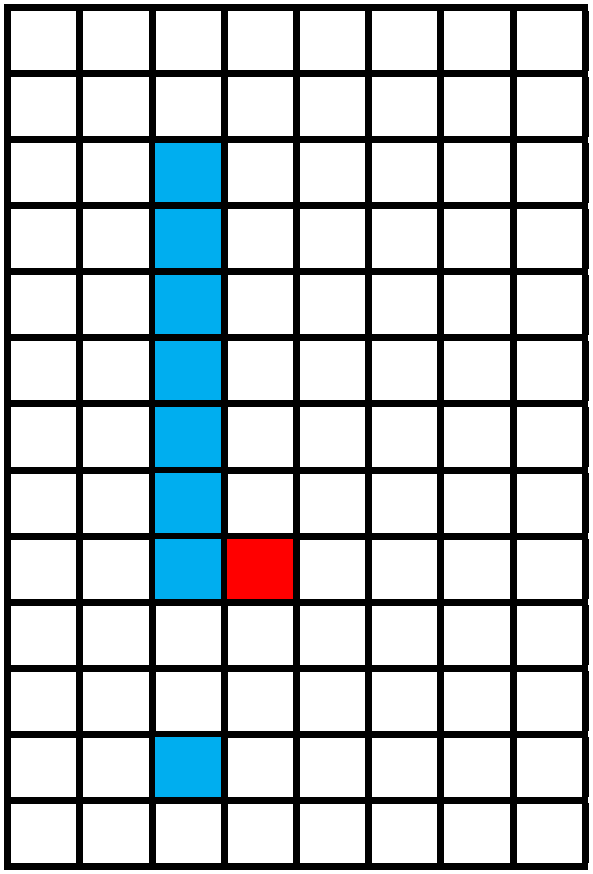
$M[i, j] =$ Minimum number of dice throws required to reach cell i from cell 0 after at most j movements in board.

Compute $M[n, n]$.

Step 3. Recurrence

$$M[i,j] = \begin{cases} 0 & \text{if } i = 0, j \in [0,n], \\ \infty & \text{if } i \in [1,n], j = 0, \\ 1 & \text{if } i \in [1,6], j \in [1,n], \\ \min \begin{pmatrix} M[i,j-1], \\ 1 + \min_{k \in [i-6, i-1];\ b[k]=k}(M[k,j-1]), \\ M[r[i], j-1] \end{pmatrix} & \text{if } i \in [7,n], j \in [1,n] \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{SnakeAndLadders}(b[1..n], r[1..n])$

Input: Board $b[1..n]$, reachable array $r[1..n]$ with snakes and ladders.
Output: Minimum number of dice throws needed to reach n.

```
M[0, 0..n] ← 0; M[1..n, 0] ← ∞
for j ← 1 to n do
  for i ← 1 to 6 do
    M[i, j] ← 1
  for i ← 7 to n do
    M[i, j] ← min(M[i, j − 1], M[r[i], j − 1])
    for k ← 1 to 6 do
      if b[i − k] = i − k then
        M[i, j] ← min(M[i, j], 1 + M[i − k, j − 1])
return M[n, n]
```

Step 6. Table

i	1	2	3	4	5	6	7	8	9
$b[i]$	1	2	3	9	7	6	7	8	9
$r[i]$	1	2	3	4	5	6	5	8	4

$M[i,j]$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	∞	1	1	1	1	1	1	1	1	1
2	∞	1	1	1	1	1	1	1	1	1
3	∞	1	1	1	1	1	1	1	1	1
4	∞	1	1	1	1	1	1	1	1	1
5	∞	1	1	1	1	1	1	1	1	1
6	∞	1	1	1	1	1	1	1	1	1
7	∞	∞	1	1	1	1	1	1	1	1
8	∞	∞	2	2	2	2	2	2	2	2
9	∞	∞	1	1	1	1	1	1	1	1

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Step 2. Subproblem

$M[i] =$ Minimum number of dice throws required to reach cell i from cell 0.

Compute $M[n]$.

Step 3. Recurrence

$$M^t[i] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0, \\ \min \begin{pmatrix} M^{t-1}[i], \\ 1 + \min_{k \in [i-6, i-1];\ k \geq 0;\ b[k]=k} (M^{t-1}[k]), \\ M^{t-1}[r[i]] \end{pmatrix} & \text{if } i \neq 0 \end{array} \right\}$$

Step 4. Dependency

Step 5. Algorithm

$\textsc{SnakeAndLadders}(b[1..n], r[1..n])$

Input: Board $b[1..n]$, reachable array $r[1..n]$ with snakes and ladders.
Output: Minimum number of dice throws needed to reach n.

```
M[0] ← 0; M[1..min(n,6)] ← 1; M[7..n] ← ∞
minheap H ← {(1,1)..(1,min(n,6))}                ▷ pair of (throws,cell)
while heap H is nonempty do
  (d,j) ← H.DeleteMin()                          ▷ cell with minimum throws
  if d ≤ M[j] then
    if b[j] = j then
      for i ← j+1 to min(j+6,n) do
        if M[i] > d+1 then
          M[i] ← d+1
          H.Insert(d+1,i)
    else if b[j] ≠ j then
      if M[b[j]] > d then
        M[b[j]] ← d
        H.Insert(d,b[j])
return M[n]
```

Step 6. Table

i	1	2	3	4	5	6	7	8	9
$b[i]$	1	2	3	9	7	6	7	8	9
$r[i]$	1	2	3	4	5	6	5	8	4
$M[i]$	1	1	1	1	1	1	1	2	1

Step 7. Complexity

$$\text{Time} \in \Theta\left(n \log n\right), \text{Space} \in \Theta\left(n\right)$$

Cutting Cloth HOME

Step 1. Problem

- [Link] We are given a rectangular piece of cloth with dimensions $m \times n$ where m and n are positive integers, and a list of k cloth products such that $\ell[i], w[i], c[i]$ represent the length, width, and cost, respectively, of cloth product i, for $i \in [1, k]$. A machine can cut a cloth in two pieces, either vertically or horizontally.
- Calculate the maximum profit that can be made if there is no limit to the maximum number of copies of a particular item and there is no minimum number of copies of a product required.

Step 2. Subproblem

$C[i, j] =$ Maximum profit that can be made
from a cloth of length i and width j.

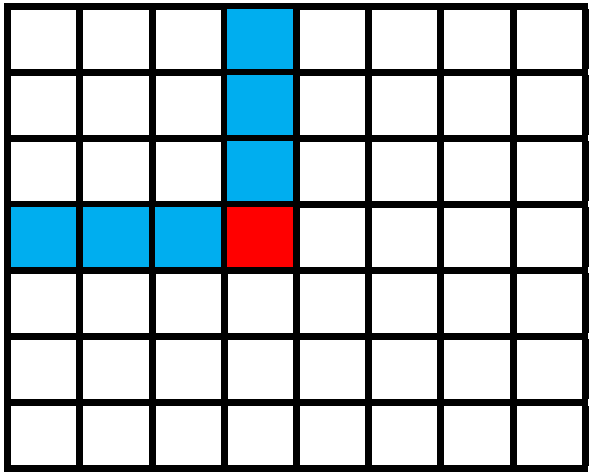
Compute $C[m, n]$.

Step 3. Recurrence

$$
\begin{aligned}
R[i,j] &= \text{Maximum cost of a product of length } i \text{ and width } j \\
&= \max\left(\max_{p\in[1,k],\ell[p]=i,w[p]=j} c[p], 0\right)
\end{aligned}
$$

$$
C[i,j] = \left\{\begin{array}{ll}
\max\Big(0, R[i,j]\Big) & \text{if } i=1 \text{ or } j=1, \\
\max\left\{\begin{array}{l}
\max_{p\in[1,i-1]}\Big(C[p,j], C[i-p,j]\Big), \\
\max_{p\in[1,j-1]}\Big(C[i,p], C[i,j-p]\Big), \\
R[i,j]
\end{array}\right\} & \text{if } i\in[2,m], j\in[2,n].
\end{array}\right\}
$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{CuttingCloth}(\ell[1..k], w[1..k], c[1..k], m, n)$

Input: Arrays $\ell[1..k]$, $w[1..k]$, $c[1..k]$ represent length, width, and cost, of k cloth products. m and n are the length and width of the cloth.
Output: Maximum profit that can be made.

```
R[1..m, 1..n] ← 0
for p ← 1 to k do R[ℓ[p], w[p]] ← max(R[ℓ[p], w[p]], c[p])
for i ← 1 to m do C[i, 1] ← max(0, R[i, 1])
for i ← 1 to n do C[1, i] ← max(0, R[1, i])
for i ← 2 to m do
  for j ← 2 to n do
    C[i, j] ← R[i, j]
    for p ← 1 to i − 1 do C[i, j] ← max(C[i, j], C[p, j] + C[i − p, j])
    for p ← 1 to j − 1 do C[i, j] ← max(C[i, j], C[i, p] + C[i, j − p])
return C[m, n]
```

Step 6. Table

i	1	2	3	4
$\ell[i]$	2	3	4	1
$w[i]$	1	3	2	4
$c[i]$	4	2	5	4

$C[i,j]$	1	2	3	4	5	6	7	8
1	0	0	0	3	3	3	3	6
2	4	8	12	16	20	24	28	32
3	4	8	12	19	23	27	31	38
4	8	16	24	32	40	48	56	64
5	8	16	24	35	43	51	59	70
6	12	24	36	48	60	72	84	96
7	12	24	36	51	63	75	87	102
8	16	32	48	64	80	96	112	128

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn(m+n)\right), \text{Space} \in \Theta\left(mn\right)$$

Arranging Balls HOME

Step 1. Problem

- [Link] Given $n_r \geq 0$ red balls, $n_g \geq 0$ green balls, and $n_b \geq 0$ blue balls, count the number of ways to arrange the balls in a straight line such that no two adjacent balls are of the same color.
- Suppose $n_r = 2$, $n_g = 1$, $n_b = 2$, then the answer is 12. The possible arrangements are: $\{rgbrb, rbgrb, rbgbr, rbrgb, rbrbg, bgrbr, brgbr, brgrb, brbgr, brbrg, gbrbr, grbrb\}$.

Step 2. Subproblem

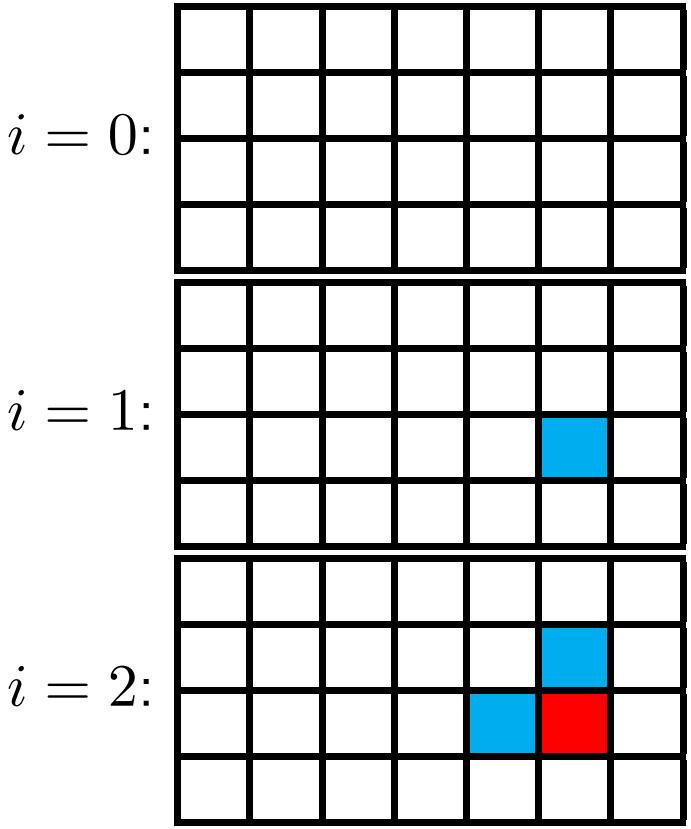
$C[i, j, k, \ell] = \#$Ways to arrange i red balls, j green balls, and k blue balls, such that the last ball is of color ℓ and no two adjacent balls have the same color.

Compute $C[n_r, n_g, n_b, r] + C[n_r, n_g, n_b, g] + C[n_r, n_g, n_b, b]$.

Step 3. Recurrence

$$C[i,j,k,\ell] = \left\{ \begin{array}{ll} 0 & \text{if } i+j+k < 1, \\ 1 & \text{if } i+j+k = 1, \\ C[i-1,j,k,g] + C[i-1,j,k,b] & \text{if } i+j+k > 1, \ell = r, \\ C[i,j-1,k,r] + C[i,j-1,k,b] & \text{if } i+j+k > 1, \ell = g, \\ C[i,j,k-1,r] + C[i,j,k-1,g] & \text{if } i+j+k > 1, \ell = b. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{ArrangeBalls}(n_r, n_g, n_b)$

Input: Number of balls of color red n_r, green n_g and blue n_b

Output: Number of ways in which we can arrange the balls such that no two balls of the same color are adjacent.

```
C[0,0,0,r] ← 0; C[0,0,0,g] ← 0; C[0,0,0,b] ← 0
C[1,0,0,r] ← 1; C[1,0,0,g] ← 0; C[1,0,0,b] ← 0
C[0,1,0,r] ← 0; C[0,1,0,g] ← 1; C[0,1,0,b] ← 0
C[0,0,1,r] ← 0; C[0,0,1,g] ← 0; C[0,0,1,b] ← 1
for i ← 0 to n_r do
  for j ← 0 to n_g do
    for k ← 0 to n_b do
      if i + j + k > 1 then
        C[i,j,k,r] ← ⟨(i ≥ 1) ? (C[i−1,j,k,g] + C[i−1,j,k,b]) : 0⟩
        C[i,j,k,g] ← ⟨(j ≥ 1) ? (C[i,j−1,k,r] + C[i,j−1,k,b]) : 0⟩
        C[i,j,k,b] ← ⟨(k ≥ 1) ? (C[i,j,k−1,r] + C[i,j,k−1,g]) : 0⟩
return (C[n_r,n_g,n_b,r] + C[n_r,n_g,n_b,g] + C[n_r,n_g,n_b,b])
```

Step 6 Table

$n_r = 2$ red balls, $n_g = 1$ green ball, $n_b = 2$ blue balls

$i = 0$	$k = 0$	$k = 1$	$k = 2$
$j = 0$	[0, 0, 0]	[0, 0, 1]	[0, 0, 0]
$j = 1$	[0, 1, 0]	[0, 1, 1]	[0, 0, 1]

$i = 1$	$k = 0$	$k = 1$	$k = 2$
$j = 0$	[1, 0, 0]	[1, 0, 1]	[0, 0, 1]
$j = 1$	[1, 1, 0]	[2, 2, 2]	[1, 1, 4]

$i = 2$	$k = 0$	$k = 1$	$k = 2$
$j = 0$	[0, 0, 0]	[1, 0, 0]	[1, 0, 1]
$j = 1$	[1, 0, 0]	[4, 1, 1]	[5, 2, 5]

Total number of ways to arrange the balls is 12.

Step 7. Complexity

$$\text{Time} \in \Theta\left(n_r n_g n_b\right), \text{Space} \in \Theta\left(n_r n_g n_b\right)$$

Word Wrap HOME

Step 1. Problem

- [Link] Given a sequence of words $w[1..n]$ and a limit on the number of characters k that can be put in one line, put line breaks such that the total cost of putting breaks is minimized.
- The total cost is the sum of costs for all lines. The cost of putting breaks in a line is the squared of the number of character spaces left unutilized in the line, except for the last line.
- Suppose $k = 6, w[1..8] = [abc, bb, ac, cabab, cc, bcd, aba, cab]$.

	1	2	3	4	5	6	cost
line 1	a	b	c		b	b	0
line 2	a	c	■	■	■	■	16
line 3	c	a	b	a	b	■	1
line 4	c	c		b	c	d	0
line 5	a	b	a	■	■	■	9
line 6	c	a	b				0

Suboptimal Greedy (Cost = 26)

	1	2	3	4	5	6	cost
line 1	a	b	c	■	■	■	9
line 2	b	b		a	c	■	1
line 3	c	a	b	a	b	■	1
line 4	c	c		b	c	d	0
line 5	a	b	a	■	■	■	9
line 6	c	a	b				0

Optimal DP (Cost = 20)

Step 2. Subproblem

$C[j] =$ Cost of putting words $w[1..j]$ in lines each of length k.

Compute $C[n]$.

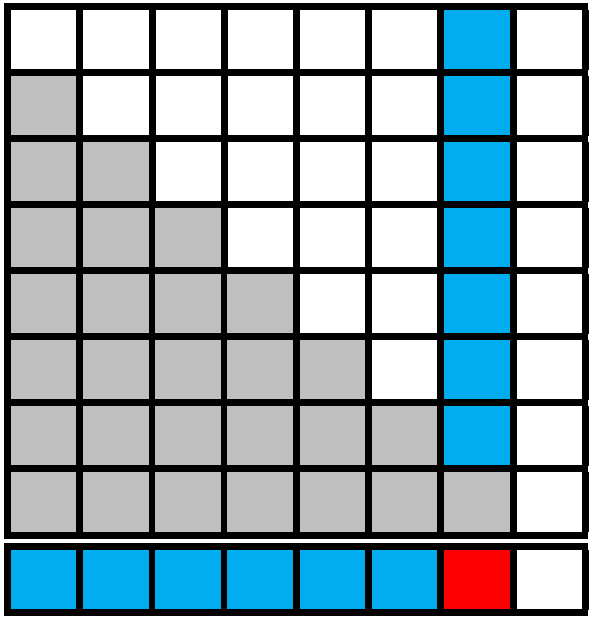
Step 3. Recurrence

$c[i,j]$ = Cost of putting words $w[i..j]$ in a single line.

($c[i,j]$ is ∞ if sum of the lengths of words $w[i..j]$ is greater than k.)

$$C[j] = \left\{ \begin{array}{ll} 0 & \text{if } j = 0, \\ \min \left\{ \begin{array}{l} C[0] + c[1,j], \\ C[1] + c[2,j], \\ \dots \\ C[j-1] + c[j,j] \end{array} \right\} & \text{if } j \in [1,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{WordWrap}(w[1..n], k)$

Input: Array of words $w[1..n]$, line width k characters.
Output: Minimum cost $C[n]$ of putting words $w[1..n]$ in lines.
If the length of any word is greater than k, then there is no solution.

```
for i ← 1 to n do ℓ[i] ← w[i].length
for i ← 1 to n do
  width ← k
  for j ← i to n do
    if i ≠ j then c[i, j] ← width − 1 − ℓ[j]
    else c[i, j] ← width − ℓ[j]
    width ← c[i, j]
    if c[i, j] < 0 then c[i, j] ← ∞
    else if j = n and c[i, j] > 0 then c[i, j] ← 0
    else c[i, j] ← c[i, j] × c[i, j]
. . . . . .
C[0] ← 0
for j ← 1 to n do
  minimum ← ∞
  for i ← 1 to j do minimum ← min(minimum, C[i − 1] + c[i, j])
  C[j] ← minimum
return C[n]
```

Step 6. Table

$$k = 6, w[1..8] = [abc, bb, ac, cabab, cc, bcd, aba, cab]$$

$c[i,j]$	1	2	3	4	5	6	7	8
1	9	0	∞	∞	∞	∞	∞	∞
2		16	1	∞	∞	∞	∞	∞
3			16	∞	∞	∞	∞	∞
4				1	∞	∞	∞	∞
5					16	0	∞	∞
6						9	∞	∞
7							9	∞
8								0

j	0	1	2	3	4	5	6	7	8
$C[j]$	0	9	0	10	11	27	11	20	20

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Difficulty of Job Schedule HOME

Step 1. Problem

- [Link] Given n jobs with difficulty $d[1..n]$ and k days ($k \in [1, n]$), which needs to be attended in the given order and at least one job needs to be completed on any given day. In other words, the n jobs must be partitioned in order among the k days.
- Difficulty on a day is the maximum difficulty of job completed on that day.
- The difficulty of a job schedule is the sum of difficulties of each day of the k days, find the minimum difficulty of a job schedule.

Step 2. Subproblem

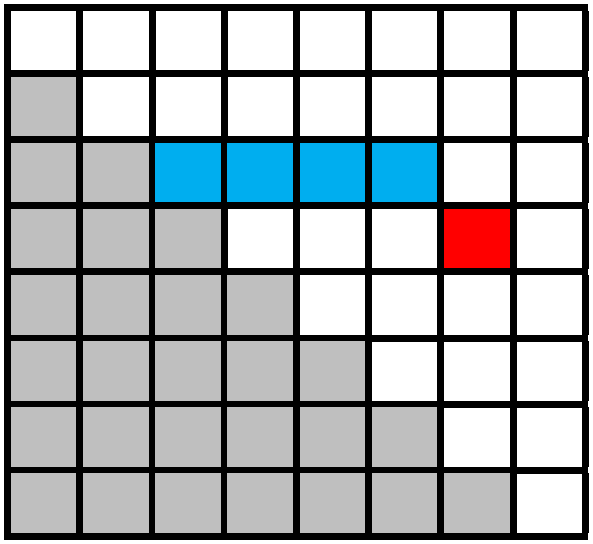
$M[i, j] =$ Minimum difficulty to finish first j jobs in i days.

Compute $M[k, n]$.

Step 3. Recurrence

$$M[i,j] = \left\{ \begin{array}{ll} 0 & \text{if } i = 0 \text{ or } j = 0 \text{ or } i > j, \\ \min \begin{pmatrix} M[i-1, j-1] + \max(d[j..j]), \\ M[i-1, j-2] + \max(d[j-1..j]), \\ \dots \\ M[i-1, i-1] + \max(d[i-1..j]) \end{pmatrix} & \text{if } i \in [1,k], j \in [1,n], i \leq j. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{MinimumDifficultyJobSchedule}(d[1..m, 1..n], k)$

Input: Jobs with difficulty $d[1..n]$ and k days.
Output: Minimum difficulty of job schedule using k days and n jobs.

```
for i ← 0 to k do M[i, 0] ← 0
for j ← 1 to n do M[0, j] ← 0
for i ← 1 to k do
  for j ← 1 to i − 1 do M[i, j] ← 0
  for j ← i to n do
    M[i, j] ← ∞
    m ← −∞
    for ℓ ← j to i do
      m ← max(m, d[ℓ])
      M[i, j] ← min(M[i, j], M[i − 1, ℓ − 1] + m)
return M[k, n]
```

Step 6. Table

$d[1..6] = \{7, 1, 7, 1, 7, 1\}$ and $k = 3$

$M[i,j]$	0 ($d[0] = \emptyset$)	1 ($d[1] = 7$)	2 ($d[2] = 1$)	3 ($d[3] = 7$)	4 ($d[4] = 1$)	5 ($d[5] = 7$)	6 ($d[6] = 1$)
0 days	0	0	0	0	0	0	0
1 day	0	7	7	7	7	7	7
2 days	0	0	8	14	8	14	8
3 days	0	0	0	15	15	15	15

Step 7. Complexity

$$\text{Time} \in \Theta\left(kn^2\right), \text{Space} \in \Theta\left(kn\right)$$

Wine Sales HOME

Step 1. Problem

- [Link] Given n wine bottles in a row with $c[1..n]$ representing the cost of each wine. Each year only the first or the last wine in the row can be sold. The price of wines increases over time. On the yth year, the profit from the ith wine will be $y \times c[i]$. Calculate the maximum profit from the sale of all wines.
- When $c[1..5] = [2, 49, 51, 1, 50]$, the maximum profit is \$556.

c[1]	c[2]	c[3]	c[4]	c[5]	gain	profit
2	49	51	1	50	1×2	2
	49	51	1	50	2×49	100
		51	1	50	3×50	250
		51	1		4×1	254
		51			5×51	509

Suboptimal Greedy

(Profit = 509)

c[1]	c[2]	c[3]	c[4]	c[5]	gain	profit
2	49	51	1	50	1×2	2
	49	51	1	50	2×50	102
	49	51	1		3×1	105
	49	51			4×49	301
		51			5×51	556

Optimal DP

(Profit = 556)

Step 2. Subproblem

$P[i, j] =$ Maximum profit made by selling wines of cost $c[i..j]$.

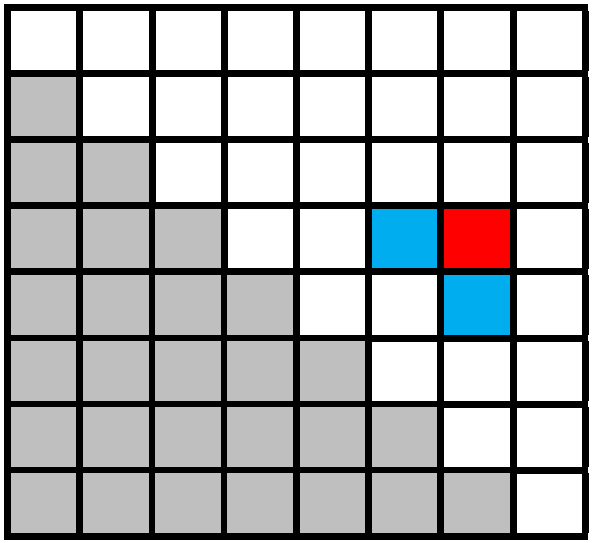
Compute $P[1, n]$.

Step 3. Recurrence

$$P[i,j] = \left\{ \begin{array}{ll} y \times c[i] & \text{if } i = j, \\ \max \left\{ \begin{array}{l} y \times c[i] + P[i+1,j], \\ P[i,j-1] + y \times c[j] \end{array} \right\} & \text{if } i \in [1, n-1], j \in [i+1, n]. \end{array} \right\}$$

where, $y =$ current year of selling wines $= n - (j - i)$.

Step 4. Dependency



Step 5. Algorithm

$\textsc{WineSales}(c[1..n])$

Input: Cost of wines $c[1..n]$ placed in a row.
Output: Maximum profit $P[1, n]$ made from selling all the wines.

```
for i ← 1 to n do
  P[i, i] ← n × c[i]
for i ← n − 1 to 1 do
  for j ← i + 1 to n do
    y ← n − j + i
    P[i, j] ← max(y × c[i] + P[i + 1, j], P[i, j − 1] + y × c[j])
return P[1, n]
```

Step 6. Table

$$c[1..5] = [2, 49, 51, 1, 50]$$

$P[i,j]$	1 ($c[1] = 2$)	2 ($c[2] = 49$)	3 ($c[3] = 51$)	4 ($c[4] = 1$)	5 ($c[5] = 50$)
1 ($c[1] = 2$)	10	253	457	459	556
2 ($c[2] = 49$)		245	451	454	554
3 ($c[3] = 51$)			255	259	409
4 ($c[4] = 1$)				5	254
5 ($c[5] = 50$)					250

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Grid Traversals HOME

Step 1. Problem

- [Link] Given a matrix $a[1..r, 1..c]$ where every cell represents points (nonnegative integers), collect maximum points using two traversals under following conditions:
- The first traversal starts from top left corner $(1, 1)$ and the second traversal starts from top right corner $(1, c)$. Both traversals should reach bottom row such that the second traversal is in the adjacent cells of first traversal, i.e., if first traversal ends at (r, j), then the second traversal ends at $(r, j + 1)$ or $(r, j - 1)$.
- From a point (i, j), we can move to $(i + 1, j - 1)$ or $(i + 1, j)$ or $(i + 1, j + 1)$.
- A traversal gets all points of a particular cell through which it passes. If one traversal has already collected points of a cell, then the other traversal gets no points if goes through that cell again.

Step 2. Subproblem

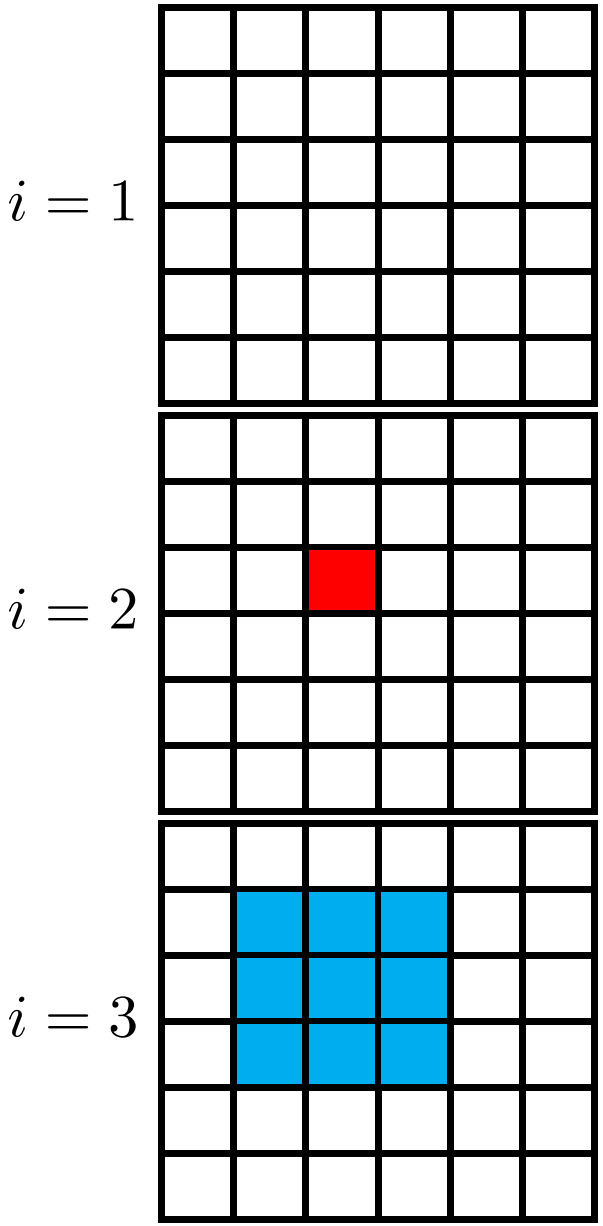
$P[i, j_1, j_2] =$ Max points that can be collected from row i to r
where first traversal is at position $[i, j_1]$
and second traversal is at position $[i, j_2]$

Compute $P[1, 1, c]$.

Step 3. Recurrence

$$P[i, j_1, j_2] = \left\{ \begin{array}{ll} -\infty & \text{if } i = r, j_2 \neq (j_1 \pm 1), \\ a[i, j_1] + a[i, j_2] & \text{if } i = r, j_2 = (j_1 \pm 1), \\ \left\{ \begin{array}{l} \max_{k_1, k_2 \in [-1,1]} (P[i+1, j_1 + k_1, j_2 + k_2]) \\ + a[i, j_1] + (a[i, j_2] \times \boxed{j_1 \neq j_2}) \end{array} \right\} & \text{if } i \in [1, r-1]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{MaxPoints}(a[1..r, 1..c])$

Input: A matrix $a[1..r, 1..c]$

Output: Maximum points collected using 2 traversals

for $j_1 \leftarrow 1$ **to** c **do**
  for $j_2 \leftarrow 1$ **to** c **do**
    if $j_2 = j_1 + 1$ **or** $j_2 = j_1 - 1$ **then** $P[r, j_1, j_2] \leftarrow a[r, j_1] + a[r, j_2]$
    else $P[r, j_1, j_2] \leftarrow -\infty$
for $i \leftarrow r - 1$ **to** 1 **do**
  for $j_1 \leftarrow 1$ **to** c **do**
    for $j_2 \leftarrow 1$ **to** c **do**

$$
\begin{aligned}
t \leftarrow\ & a[i, j_1] + a[i, j_2] \times \boxed{i \neq j} + \max(P[i+1, j_1, j_2], \\
& P[i+1, j_1 - 1, j_2] \times \boxed{j_1 > 1}, P[i+1, j_1 + 1, j_2] \times \boxed{j_1 < c}, \\
& P[i+1, j_1, j_2 - 1] \times \boxed{j_2 > 1}, P[i+1, j_1, j_2 + 1] \times \boxed{j_2 < c}, \\
& P[i+1, j_1 - 1, j_2 - 1] \times \boxed{j_1 > 1} \times \boxed{j_2 > 1}, \\
& P[i+1, j_1 - 1, j_2 + 1] \times \boxed{j_1 > 1} \times \boxed{j_2 < c}, \\
& P[i+1, j_1 + 1, j_2 - 1] \times \boxed{j_1 < c} \times \boxed{j_2 > 1}, \\
& P[i+1, j_1 + 1, j_2 + 1] \times \boxed{j_1 < c} \times \boxed{j_2 < c})
\end{aligned}
$$

return $P[1, 1, c]$

Step 6. Table

$a[i,j]$	1	2	3	4	5	6	7	8
1	8	6	8	2	1	1	9	10
2	5	7	4	3	6	4	5	2
3	1	1	9	10	5	7	6	1
4	1	1	5	10	8	2	4	3
5	1	1	3	10	9	6	8	2
6	3	6	8	3	1	11	9	10
7	2	5	16	3	6	13	5	2
8	9	1	1	5	12	3	6	1
9	1	9	1	15	10	2	4	3
10	1	1	9	10	3	6	8	2

Max points collected $= 192$.

DP table is a 3D table and hence is not shown.

Step 7. Complexity

$$\text{Time} \in \Theta\left(rc^2\right), \text{Space} \in \Theta\left(rc^2\right)$$

Sequence Alignment + Gaps HOME

Step 1. Problem

- [Link] Compute minimum cost required to transform string $x[1..m]$ into another string $y[1..n]$. When transforming a string $x[1..m]$ into another string $y[1..n]$, a sequence of consecutive deletes corresponds to a gap in X, and a sequence of consecutive inserts corresponds to a gap in Y. The cost of such a gap is not necessarily equal to the sum of the costs of each individual deletion (or insertion) in that gap. In order to handle this general case two new cost functions w and w' are defined, where $w(p,q)(0 \leq p < q \leq m)$ is the cost of deleting $x_{p+1}..x_q$ from X, and $w'(p,q)(0 \leq p < q \leq n)$ is the cost of inserting $y_{p+1}..y_q$ into X. The substitution function $s(x_i, y_j)$ is the same as that of the standard edit distance problem.

Step 2. Subproblem

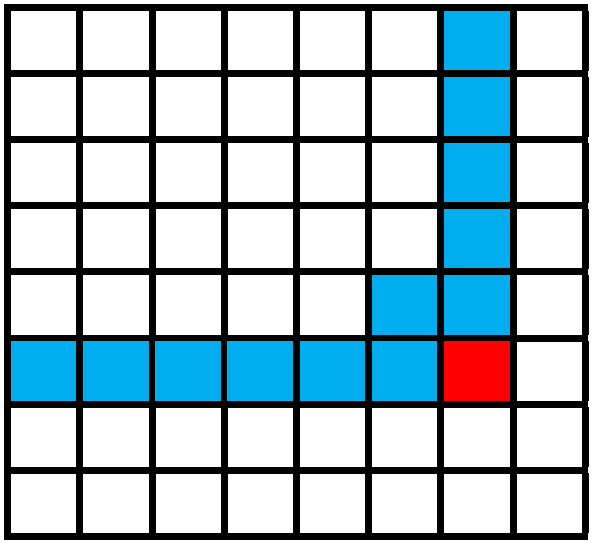
$G[i, j] =$ Minimum cost of transforming string $x[1..i]$ into $y[1..j]$ using costs of insertion, deletion, substitution and gap penalty.

Compute $G[m, n]$.

Step 3. Recurrence

$$G[i,j] = \left\{ \begin{array}{ll} 0 & \text{if } i=0, j=0, \\ w(0,j) & \text{if } i=0, j\in[1,n], \\ w'(0,i) & \text{if } i\in[1,m], j=0, \\ \min \left\{ \begin{array}{l} G[i-1,j-1]+s(x_i,y_j), \\ \min_{q\in[0,j-1]}(G[i,q]+w(q,j)) \\ \min_{p\in[0,i-1]}(G[p,j]+w'(p,i)) \end{array} \right\} & \text{if } i\in[1,m], j\in[1,n]. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{SequenceAlignmentWithGaps}(x[1..m], y[1..n])$

Input: Strings $x[1..m]$, $y[1..n]$.
Output: Minimum cost required to transfrom string $x[1..m]$ to string $y[1..n]$.

```
G[0,0] ← 0
for j ← 1 to n do G[0,j] ← w(0,j)
for i ← 1 to m do G[i,0] ← w'(0,i)
for i ← 1 to m do
  for j ← 1 to n do
    G[i,j] ← G[i−1,j−1] + s(x[i],y[j])
    for k ← 0 to j−1 do
      G[i,j] ← min(G[i,j], G[i,k] + w(k,j))
    for k ← 0 to i−1 do
      G[i,j] ← min(G[i,j], G[k,j] + w'(k,i))
return G[m,n]
```

Step 6. Table

$w[k,j]$	0	1	2	3	4	5	6	7
0		5	4	3	2	4	7	6
1			7	5	2	6	4	2
2				1	2	1	2	3
3					8	4	2	1
4						4	2	5
5							3	2
6								2
7								

$w'[k,i]$	0	1	2	3	4
0		2	3	5	6
1			4	7	8
2				9	10
3					5
4					

$G[i,j]$	0 ($y[0] = \phi$)	1 ($y[1] = a$)	2 ($y[2] = t$)	3 ($y[3] = g$)	4 ($y[4] = g$)	5 ($y[5] = c$)	6 ($y[6] = c$)	7 ($y[7] = t$)
0 ($x[0] = \phi$)	0	5	4	3	2	4	7	6
1 ($x[1] = g$)	2	1	6	5	3	5	5	3
2 ($x[2] = a$)	3	4	3	4	5	4	5	5
3 ($x[3] = c$)	5	7	8	7	7	9	8	8
4 ($x[4] = t$)	6	9	10	9	8	10	10	10

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn \max(m,n)\right), \text{Space} \in \Theta\left(mn\right)$$

Longest Palindromic Substring HOME

Step 1. Problem

- [Link] Given a string $s[0..n-1]$, find the length of the longest substring in it which is a palindrome.
- Suppose $s = aaaabbaa$, then the output should be 6 as $aabbaa$ is the longest palindromic substring in it.

Step 2. Subproblem

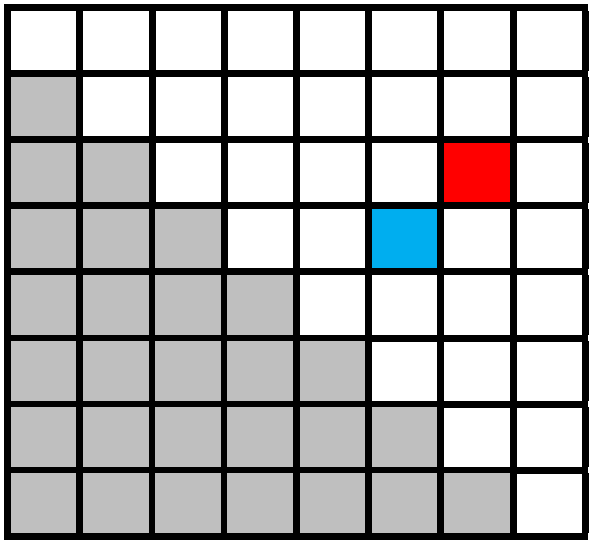
$P[i, j] =$ Boolean value to represent if the input substring $s[i..j]$ is a palindrome or not.

Compute $\max_{i,j \in [0,n-1]}$ and $P[i,j]=true(j - i + 1)$.

Step 3. Recurrence

$$P[i,j] = \left\{ \begin{array}{ll} true & \text{if } i = j, \\ false & \text{if } i > j, \\ \boxed{s[i] = s[j]} & \text{if } i = j-1, \\ \boxed{s[i] = s[j] \text{ and } P[i+1, j-1]} & \text{if } i < j-1. \end{array} \right\}$$

Step 4. Dependency



Step 5. Algorithm

LongestPalindromicSubstring($s[0..n-1]$)

Input: String $s[0..n-1]$
Output: Maximum length $maxlength$ of the longest palindromic substring of string $s[0..n-1]$

```
maxlength ← 1
for i ← 0 to n − 1 do P[i, i] ← true
for i ← 0 to n − 2 do
  if s[i] = s[i + 1] then
    P[i, i + 1] ← true; maxlength ← 2
for ℓ ← 3 to n do
  for i ← 0 to n − ℓ do
    j ← i + ℓ − 1
    if P[i + 1, j − 1] = true and s[i] = s[j] then
      P[i, j] ← true; maxlength ← max(maxlength, ℓ)
return maxlength
```

Step 6. Table

String $s[0..7] = aaaabbaa$

$P[i,j]$	0 ($s[0] = a$)	1 ($s[1] = a$)	2 ($s[2] = a$)	3 ($s[3] = a$)	4 ($s[4] = b$)	5 ($s[5] = b$)	6 ($s[6] = a$)	7 ($s[7] = a$)
0 ($s[0] = a$)	T	T	T	T	F	F	F	F
1 ($s[1] = a$)		T	T	T	F	F	F	F
2 ($s[2] = a$)			T	T	F	F	F	T
3 ($s[3] = a$)				T	F	F	T	F
4 ($s[4] = b$)					T	T	F	F
5 ($s[5] = b$)						T	F	F
6 ($s[6] = a$)							T	T
7 ($s[7] = a$)								T

Output $= 7 - 2 + 1 = 6$

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Longest Palindromic Subsequence HOME

Step 1. Problem

- [Link] Given a string $s[0..n-1]$, find the length of the longest palindromic subsequence in it.
- Example: String $s = ababbbaabb$. Answer: 7.
  Because, $babbbab$ is the longest palindromic subsequence.

Step 2. Subproblem

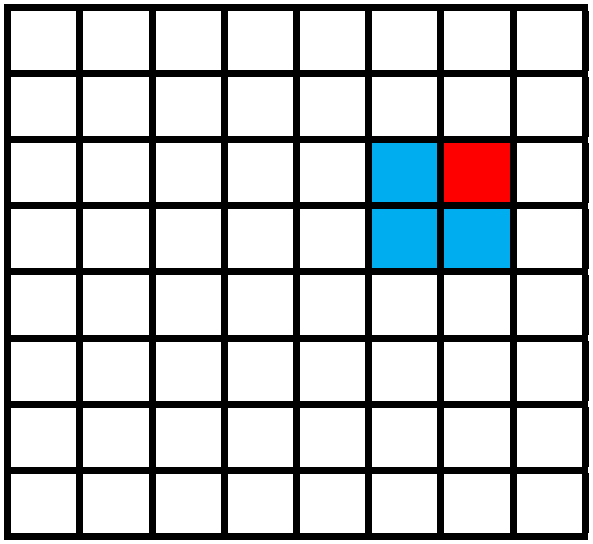
$L[i, j] =$ Length of the longest palindromic subsequence in string $s[i..j]$.

Compute $L[0, n-1]$.

Step 3. Recurrence

$$L[i,j] = \begin{cases} 0 & \text{if } i > j, \\ 1 & \text{if } i = j, \\ \left\{ \begin{array}{l} (2 + L[i+1, j-1]) \times \boxed{s[i] = s[j]} + \\ \max\left(L[i, j-1], L[i+1, j]\right)) \times \boxed{s[i] \neq s[j]} \end{array} \right\} & \text{if } i < j. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

LongestPalindromicSubsequence($s[0..n-1]$)

Input: String $s[0..n-1]$
Output: Length $L[0, n-1]$ of the longest palindromic subsequence of string $s[0..n-1]$

```
for i ← 0 to n − 1 do L[i, i] ← 1
for ℓ ← 2 to n do
  for i ← 0 to n − ℓ do
    j ← i + ℓ − 1
    if s[i] = s[j] then
      L[i, j] ← 2
      if j > i + 1 then L[i, j] ← L[i, j] + L[i + 1, j − 1]
    else if s[i] ≠ s[j] then
      L[i, j] ← max(L[i, j − 1], L[i + 1, j])
return L[0, n − 1]
```

Step 6. Table

String $s[0..9] = ababbbaabb$

$L[i,j]$	0	1	2	3	4	5	6	7	8	9
0	1	1	3	3	3	4	6	7	7	7
1		1	1	3	3	4	5	5	7	7
2			1	1	2	3	5	5	5	6
3				1	2	3	3	3	4	6
4					1	2	2	2	4	6
5						1	1	2	4	4
6							1	2	2	2
7								1	1	2
8									1	2
9										1

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n\right)$$

Longest Bitonic Subsequence HOME

Step 1. Problem

- [Link] Given an array $a[0..n-1]$ containing n positive integers, find the length of the longest bitonic subsequence.
- Suppose input array is $a = [1, 11, 2, 10, 4, 5, 2, 1]$, then the output is 6. A longest bitonic subsequence of length 6 is $[1, 2, 10, 4, 2, 1]$.

Step 2. Subproblem

$LIS[i] =$ Length of the longest increasing subsequence in $a[0..i]$.
$LDS[i] =$ Length of longest decreasing subsequence in $a[i..n-1]$.

$LIS[i] + LDS[i] - 1 =$ Length of the longest bitonic subsequence in $a[0..n-1]$ with peak at i.

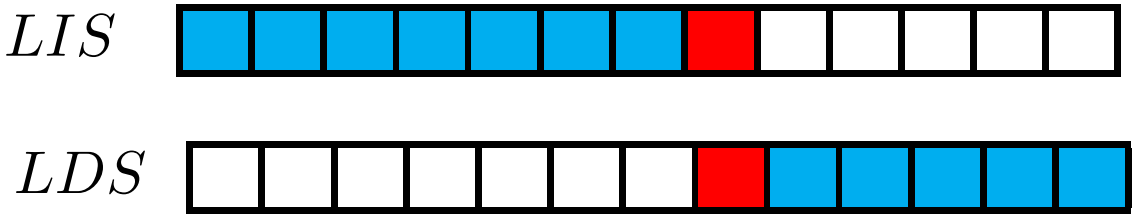
Compute $\max_{i \in [0,n-1]}(LIS[i] + LDS[i] - 1)$.

Step 3. Recurrence

$$LIS[i] = \left\{ 1 + \max \begin{cases} LIS[0] \times \boxed{A[0] < A[i]}, \\ LIS[1] \times \boxed{A[1] < A[i]}, \\ \dots \\ LIS[i-1] \times \boxed{A[i-1] < A[i]} \end{cases} \quad \text{if } i \in [0, n-1]. \right\}$$

$$LDS[i] = \left\{ 1 + \max \begin{cases} LDS[i+1] \times \boxed{A[i] > A[i+1]}, \\ LDS[i+2] \times \boxed{A[i] > A[i+2]}, \\ \dots \\ LDS[n-1] \times \boxed{A[i] > A[n-1]}, \end{cases} \quad \text{if } i \in [0, n-1]. \right\}$$

Step 4. Dependency



Step 5. Algorithm

LongestBitonicSubsequence($a[0..n-1]$)

Input: Array $a[0..n-1]$ containing n positive integers.
Output: Length of the longest bitonic subsequence of array $a[0..n-1]$.

```
for i ← 0 to n − 1 do LIS[i] ← 0; LDS[i] ← 0
for i ← 0 to n − 1 do
  for j ← 0 to i − 1 do
    if a[j] < a[i] and LIS[j] > LIS[i] then
      LIS[i] ← LIS[j]
  LIS[i] ← LIS[i] + 1
for i ← n − 1 to 0 do
  for j ← n − 1 to i + 1 do
    if a[j] < a[i] and LDS[j] > LDS[i] then
      LDS[i] ← LDS[j]
  LDS[i] ← LDS[i] + 1
result ← 0
for i ← 0 to n − 1 do
    result ← max(result, (LIS[i] + LDS[i] − 1))
return result
```

Step 6. Table

$$a[0..7] = [1, 11, 2, 10, 4, 5, 2, 1]$$

i	0	1	2	3	4	5	6	7
$LIS[i]$	1	2	2	3	3	4	2	1
$LDS[i]$	1	5	2	4	3	3	2	1

Length of the longest bitonic subsequence $= 6$

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n\right)$$

Longest Repeating Subsequence HOME

Step 1. Problem

- [Link] Given a string $s[1..n]$, find the length of the longest repeating subsequence such that the two subsequences don't have same string character at the same position, i.e., any ith character in the two subsequences shouldn't have the same index in the original string.
- Suppose $s = aabebcdd$. Then, the output should be 3. Because, abd is a longest repeating subsequence of the string.

Step 2. Subproblem

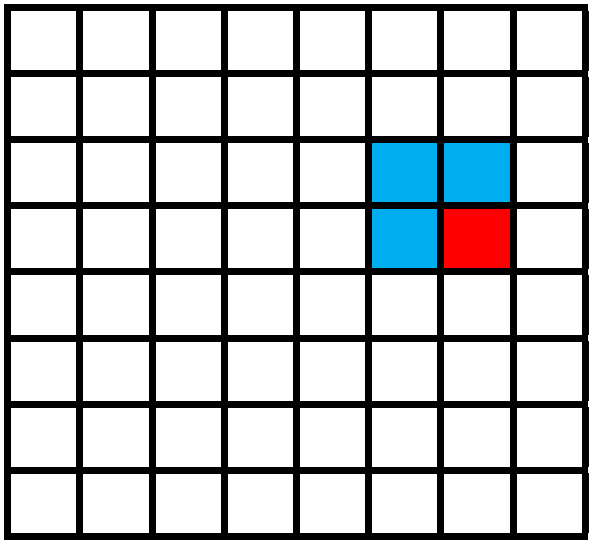
$L[i, j] =$ Length of the longest repeating subsequence between strings $s[1..i]$ and $s[1..j]$.

Compute $L[n, n]$.

Step 3. Recurrence

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \left\{ \begin{array}{l} (1 + L[i-1, j-1]) \times \boxed{s[i] = s[j] \text{ and } i \neq j} + \\ \max\left(L[i, j-1], L[i+1, j]\right)) \times \boxed{s[i] \neq s[j] \text{ or } i = j} \end{array} \right\} & \text{if } i, j \in [1, n]. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

LongestRepeatingSubsequence($s[1..n]$)

Input: String $s[1..n]$
Output: Length $L[n, n]$ of the longest repeating subsequence of s

```
for i ← 0 to n do L[i, 0] ← 0
for j ← 1 to n do L[0, j] ← 0
for i ← 1 to n do
  for j ← 1 to n do
    if s[i] = s[j] and i ≠ j then
      L[i, j] ← 1 + L[i − 1, j − 1]
    else if s[i] ≠ s[j] or i = j then
      L[i, j] ← max(L[i, j − 1], L[i + 1, j])
return L[n, n]
```

Step 6. Table

$$s[1..8] = aabebcdd$$

$L[i,j]$	0 ($s[0] = \phi$)	1 ($s[1] = a$)	2 ($s[2] = a$)	3 ($s[3] = b$)	4 ($s[4] = e$)	5 ($s[5] = b$)	6 ($s[6] = c$)	7 ($s[7] = d$)	8 ($s[8] = d$)
0 ($s[0] = \phi$)	0	0	0	0	0	0	0	0	0
1 ($s[1] = a$)	0	0	1	1	1	1	1	1	1
2 ($s[2] = a$)	0	1	1	1	1	1	1	1	1
3 ($s[3] = b$)	0	1	1	1	1	2	2	2	2
4 ($s[4] = e$)	0	1	1	1	1	2	2	2	2
5 ($s[5] = b$)	0	1	1	2	2	2	2	2	2
6 ($s[6] = c$)	0	1	1	2	2	2	2	2	2
7 ($s[7] = d$)	0	1	1	2	2	2	2	2	3
8 ($s[8] = d$)	0	1	1	2	2	2	2	3	3

Step 7. Complexity

$$\text{Time} \in \Theta\left(n^2\right), \text{Space} \in \Theta\left(n^2\right)$$

Shortest Common Supersequence HOME

Step 1. Problem

- [Link] Given two strings $x[1..m]$ and $y[1..n]$, find the length of the shortest string that has both x and y as subsequences.
- Suppose $x = ghthamb$ and $y = aggtab$ then the output should be 9 as $aghgthamb$ is the shortest string that has both x and y as subsequences.

Step 2. Subproblem

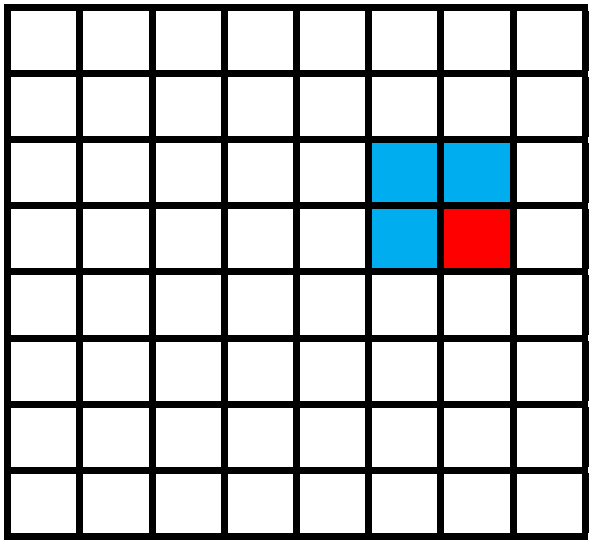
$L[i, j] =$ Length of the shortest common supersequence between strings $x[1..i]$ and $y[1..j]$.

Compute $L[m, n]$.

Step 3. Recurrence

$$L[i,j] = \begin{cases} j & \text{if } i = 0, j \in [0,n], \\ i & \text{if } i \in [1,m], j = 0, \\ \left\{ \begin{array}{l} (1 + L[i-1,j-1]) \times \boxed{x[i] = y[j]} + \\ (1 + \min(L[i-1,j], L[i,j-1])) \times \boxed{x[i] \neq y[j]} \end{array} \right\} & \text{if } i \in [1,m], j \in [1,n]. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{ShortestCommonSupersequence}(x[1..m], y[1..n])$

Input: String $x[1..m], y[1..n]$
Output: Length $L[m, n]$ of the shortest common supersequence of strings x and y

```
for j ← 0 to n do L[0, j] ← j
for i ← 0 to m do L[i, 0] ← i
for i ← 1 to m do
  for j ← 1 to n do
    if x[i] = y[j] then
      L[i, j] ← 1 + L[i − 1, j − 1]
    else if x[i] ≠ y[j] then
      L[i, j] ← 1 + min(L[i − 1, j], L[i, j − 1])
return L[m, n]
```

Step 6. Table

$$x[1..7] = ghthamb,\ y[1..6] = aggtab$$

$L[i,j]$	0 ($y[0] = \phi$)	1 ($y[1] = a$)	2 ($y[2] = g$)	3 ($Y[3] = g$)	4 ($y[4] = t$)	5 ($y[5] = a$)	6 ($y[6] = b$)
0 ($x[0] = \phi$)	0	1	2	3	4	5	6
1 ($x[1] = g$)	1	2	2	3	4	5	6
2 ($x[2] = h$)	2	3	3	4	5	6	7
3 ($x[3] = t$)	3	4	4	5	5	6	7
4 ($x[4] = h$)	4	5	5	6	6	7	8
5 ($x[5] = a$)	5	5	6	7	7	7	8
6 ($x[6] = m$)	6	6	7	8	8	8	9
7 ($x[7] = b$)	7	7	8	9	9	9	9

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(mn\right)$$

Longest Alternating Subsequence HOME

Step 1. Problem

- [Link] Given an array $a[0..n-1]$ containing n integers, find the length of the longest alternating subsequence.
- A sequence $[x_1, x_2, \ldots, x_n]$ is alternating sequence if its elements satisfy one of the following relations:
  $x_1 < x_2 > x_3 < x_4 > x_5 < \cdots$
  $x_1 > x_2 < x_3 > x_4 < x_5 > \cdots$
- Suppose input array is $a = [-4, 3, 7, -5, 0, 9, 2, -3, 8, 6]$, then the output is 7. A longest alternating subsequence of length 7 is $[-4, 3, -5, 0, -3, 8, 6]$.

Step 2. Subproblem

$I[i]$ = Length of the longest alternating subsequence in $a[0..i]$ with $a[i]$ greater than the previous element in the subsequence.

$D[i]$ = Length of the longest alternating subsequence in $a[0..i]$ with $a[i]$ less than the previous element in the subsequence.

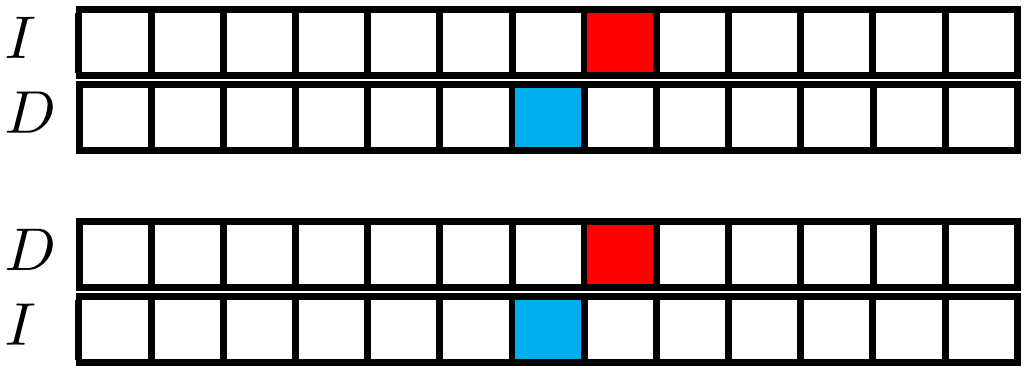
Compute $\max(I[n-1], D[n-1])$.

Step 3. Recurrence

$$I[i] = \begin{cases} 1 & \text{if } i = 0, \\ \left\{ \begin{array}{l} (1 + D[i-1]) \times \boxed{a[i] > a[i-1]} + \\ I[i-1] \times \boxed{a[i] \not> a[i-1]} \end{array} \right\} & \text{if } i \in [1, n-1]. \end{cases}$$

$$D[i] = \begin{cases} 1 & \text{if } i = 0, \\ \left\{ \begin{array}{l} (1 + I[i-1]) \times \boxed{a[i] < a[i-1]} + \\ D[i-1] \times \boxed{a[i] \not< a[i-1]} \end{array} \right\} & \text{if } i \in [1, n-1]. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{LongestAlternatingSubsequence}(a[0..n-1])$

Input: Array $a[0..n-1]$ containing n integers.
Output: Length of the longest alternating subsequence of array $a[0..n-1]$.

1. $I[0] \leftarrow 1$; $D[0] \leftarrow 1$
2. **for** $i \leftarrow 1$ **to** $n-1$ **do**
3. $\quad I[i] \leftarrow (a[i] > a[i-1])\ ?\ (1 + D[i-1]) : (I[i-1])$
4. $\quad D[i] \leftarrow (a[i] < a[i-1])\ ?\ (1 + I[i-1]) : (D[i-1])$
5. **return** $\max(I[n-1], D[n-1])$

Step 6. Table

$$a[0..9] = [-4, 3, 7, -5, 0, 9, 2, -3, 8, 6]$$

i	0	1	2	3	4	5	6	7	8	9
$I[i]$	1	2	2	2	4	4	4	4	6	6
$D[i]$	1	1	1	3	3	3	5	5	5	7

Length of the longest alternating subsequence = 7

Step 7. Complexity

$$\text{Time} \in \Theta\left(n\right), \text{Space} \in \Theta\left(n\right)$$

Count Subsequence Occurrences HOME

Step 1. Problem

- [Link] Given two strings text $t[1..m]$ and pattern $p[1..n]$, find the count of distinct occurrences of subsequence pattern p in text t.
- Suppose text $t = rabbbit$ and pattern $p = rabbit$, then the output is 3. The three ways in which pattern p can appear as subsequences in text t are: rabbbit, rabbbit, and rabbbit.

Step 2. Subproblem

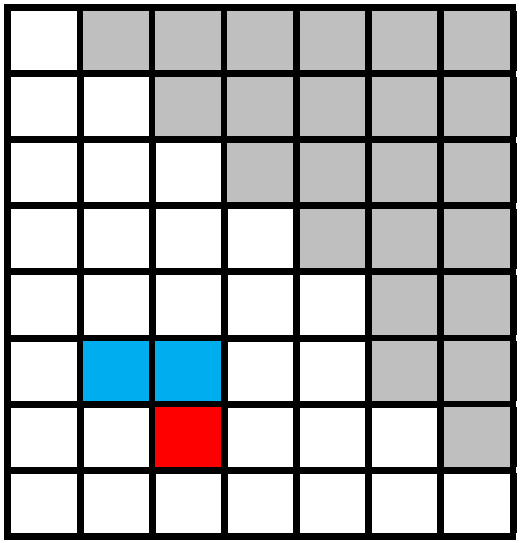
$C[i, j] =$ Count distinct occurrences of pattern $p[1..j]$ in $t[1..i]$ as a subsequence.

Compute $C[m, n]$.

Step 3. Recurrence

$$C[i,j] = \begin{cases} 1 & \text{if } i \in [0,m], j = 0, \\ 0 & \text{if } i = 0, j \in [1,n], \\ \left\{ \begin{array}{l} C[i-1,j] \times \boxed{t[i] \neq p[j]} + \\ (C[i-1,j] + C[i-1,j-1])) \times \boxed{t[i] = p[j]} \end{array} \right\} & \text{if } i \in [1,m], j \in [1,n]. \end{cases}$$

Step 4. Dependency



Step 5. Algorithm

$\textsc{CountDistinctOccurrencesAsSubsequence}(t[1..m], p[1..n])$

Input: Text $t[1..m]$ and pattern $p[1..n]$
Output: Count distinct occurrences of pattern p in text t as subsequence.

```
for i ← 0 to m do C[i, 0] ← 1
for j ← 1 to n do C[0, j] ← 0
for i ← 1 to m do
  for j ← 1 to n do
    if t[i] ≠ p[j] then
      C[i, j] ← C[i − 1, j]
    else if t[i] = p[j] then
      C[i, j] ← C[i − 1, j] + C[i − 1, j − 1]
return C[m, n]
```

Step 6. Table

$C[i,j]$	0 ($p[0] = \phi$)	1 ($p[1] = r$)	2 ($p[2] = a$)	3 ($p[3] = b$)	4 ($p[4] = b$)	5 ($p[5] = i$)	6 ($p[6] = t$)
0 ($t[0] = \phi$)	1						
1 ($t[1] = r$)	1	1					
2 ($t[2] = a$)	1	1	1				
3 ($t[3] = b$)	1	1	1	1			
4 ($t[4] = b$)	1	1	1	2	1		
5 ($t[5] = b$)	1	1	1	3	3		
6 ($t[6] = i$)	1	1	1	3	3	3	
7 ($t[7] = t$)	1	1	1	3	3	3	3

Step 7. Complexity

$$\text{Time} \in \Theta\left(mn\right), \text{Space} \in \Theta\left(mn\right)$$