Dynamic Programming

Algorithms

Definition

- A quite generic strategy that reduces the solution of a problem to the solution of similar subproblems
  - Divide and conquer:
    - Division leads to a recursion subject to the Master Theorem
    - Generate two or more subproblems
  - General dynamic programming:
    - In general, no division but a reduction of problem size
    - Leads often to super-polynomial algorithms

Definition

- Key technique:
  - **Memoization**
    - Previously obtained results are cached
  - **Tabulation**
    - Solve all previous problems and put them into a table

Usage

- Dynamic programming is very generic
  - Often, does not lead to poly-time algorithms
  - Used often when problems need to be solved even though it is known that a good scalable algorithm is unavailable
  - I.e. an NP-complete problem

Example 1
Forming sums

- Determine the number of ways we can write a number *n* as a sum of ones and twos (not using commutativity)
  - Example:

$$4 = 1 + 1 + 1 + 1$$
$$4 = 2 + 1 + 1$$
$$4 = 1 + 2 + 1$$
$$4 = 1 + 1 + 2$$
$$4 = 2 + 2$$

  - Five possibilities

Example 1
Forming sums

- Idea:
  - Sum ends with either a +1 or a +2
  - The part before sums to n-1 or n-2 respectively

Example 1
Forming sums

- Idea: The ways to write *n* are given by writing *n*-2 and *n*-1
- Number of ways for *n*: S_n
- Recurrence formula:

$$S_n = S_{n-1} + S_{n-2}$$
$$S_0 = 0$$
$$S_1 = 1$$

- Fibonacci numbers!

Example 1
Forming sums

- Extend to sums with 1, 2, 3:
  - Your turn

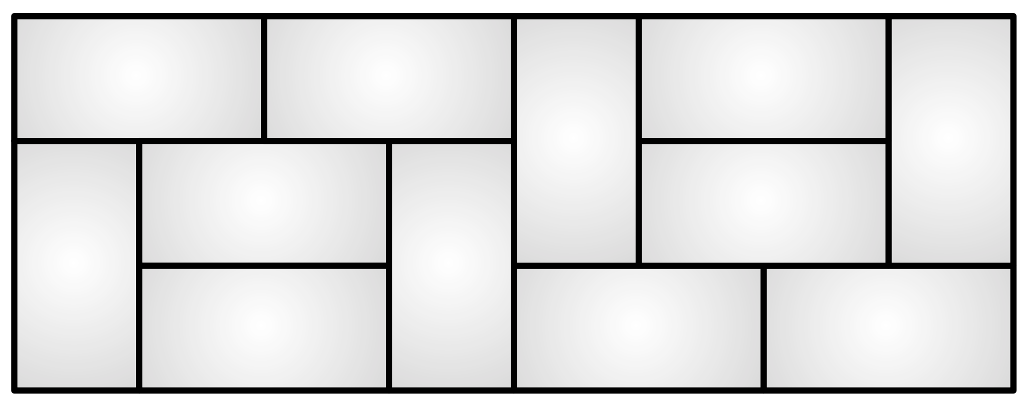
Example 1
Forming sums

- Solution

$$
\begin{aligned}
D_0 &= 0 \\
D_1 &= 1 \\
D_2 &= 2 \\
D_n &= D_{n-1} + D_{n-2} + D_{n-3}
\end{aligned}
$$

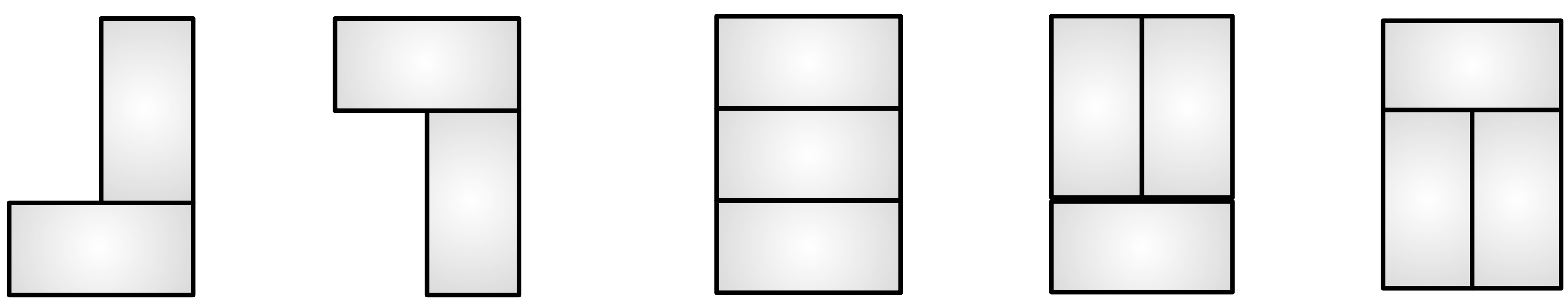
Example 2
Dominoes

- Count the number of ways in which a $3 \times n$ field can be filled with domino stones of size 2×1



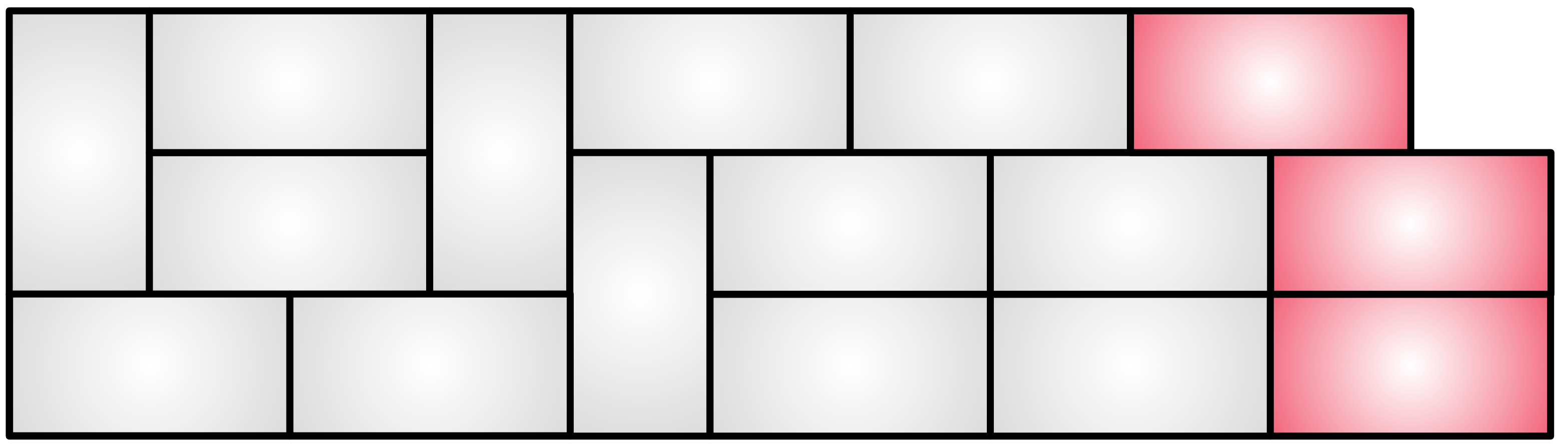
Example 2
Dominoes

- Can we reduce the problem to simpler ones?
  - T_n number of tessellations for an $3 \times n$ area
  - There is a problem for the reduction
    - We can make progress with five different stacks



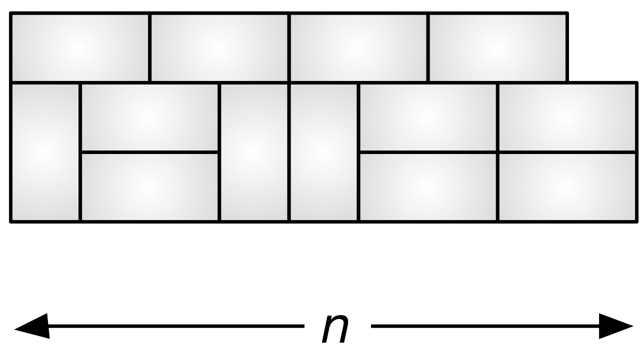
Example 2
Dominoes

- Cannot just assume that we progress by two



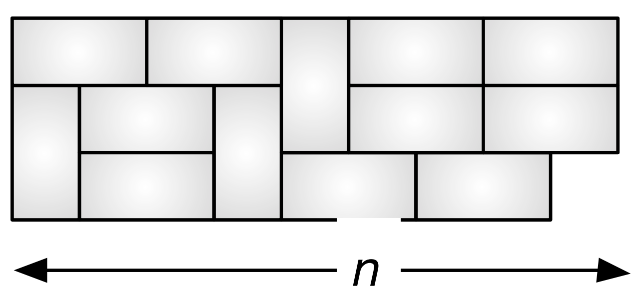
Example 2
Dominoes

- Need to introduce two more shapes



A_n

Number of tesse
of this type

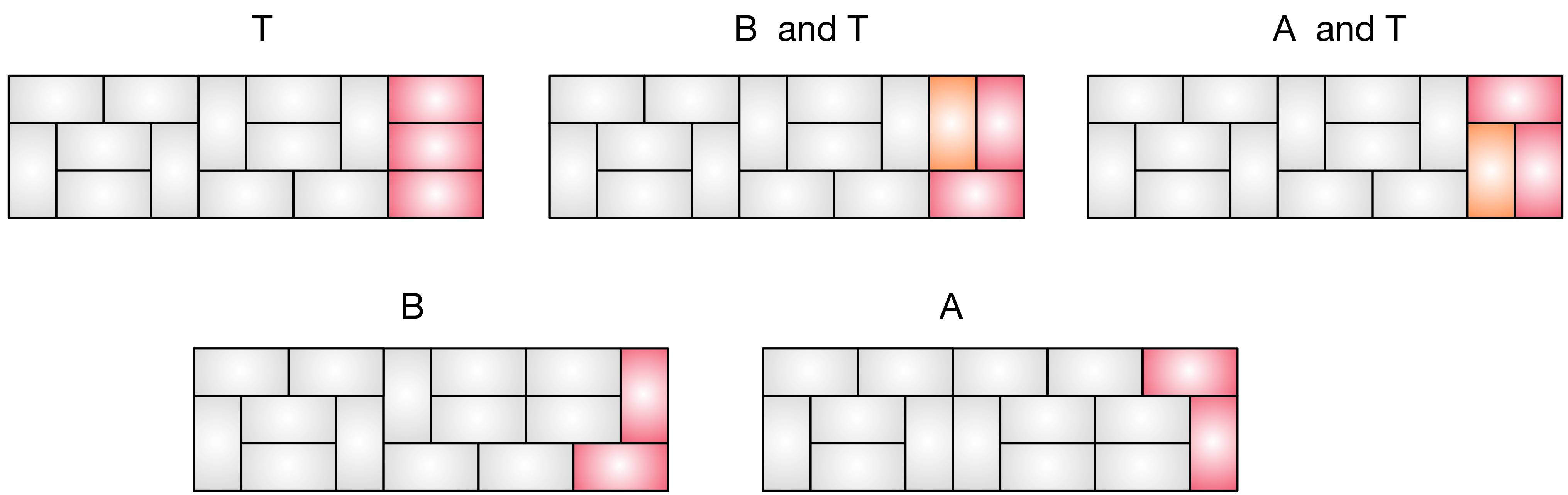


B_n

Number of tesse
of this type

Example 2
Dominoes

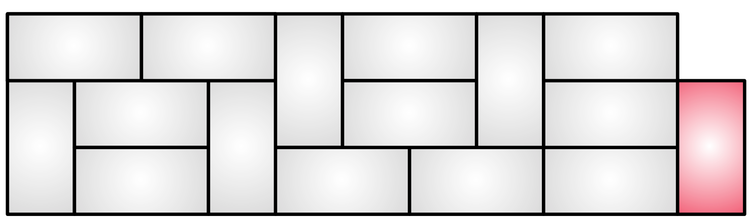
- Need recurrence for all three
- T can be generated from a T, a B and an A



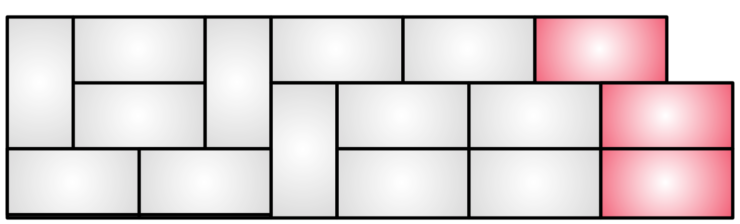
$$T_n = A_{n-1} + B_{n-1} + T_{n-2}$$

Example 2
Dominoes

- To generate a type A there are only two possibilities

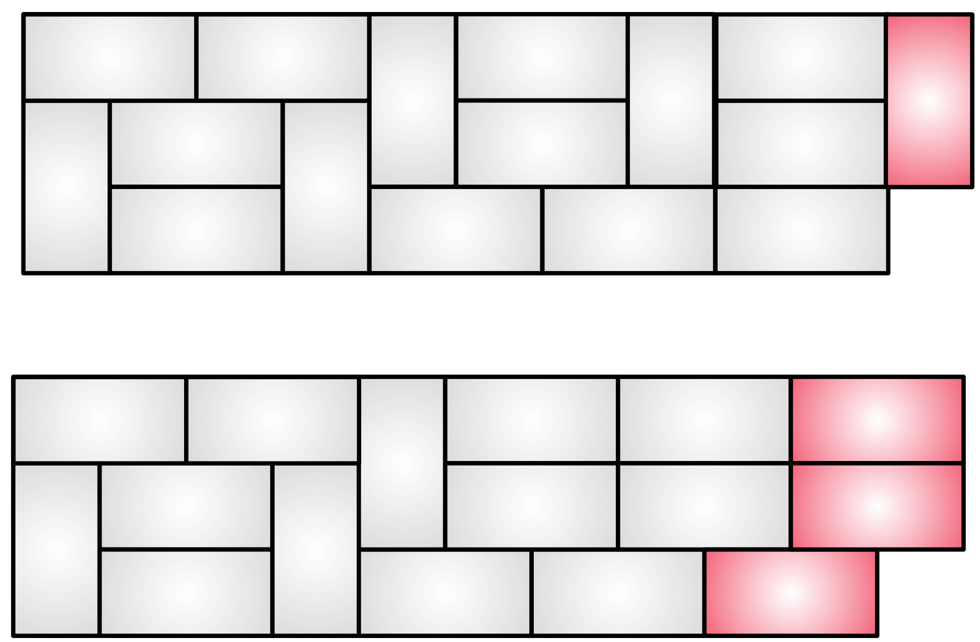


$$A_n = A_{n-2} + T_{n-1}$$



Example 2
Dominoes

- Similarly: Type B can be generated from a Type B and a Type T



$$B_n = B_{n-2} + T_{n-1}$$

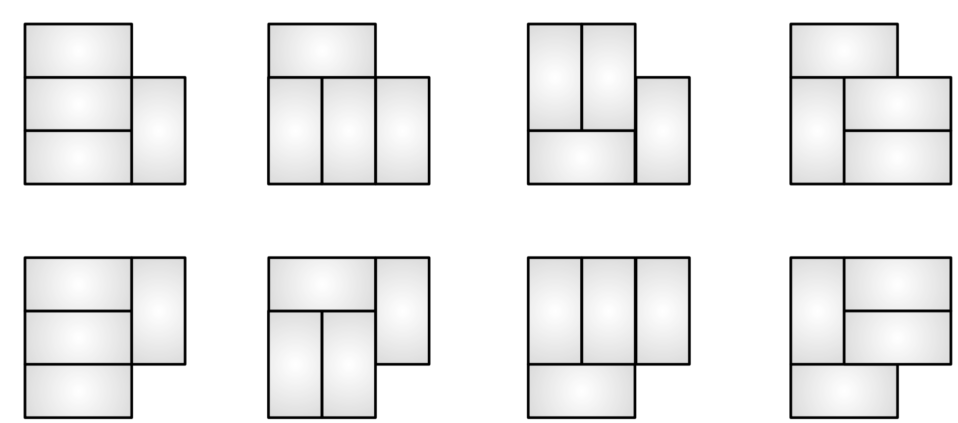
Example 2
Dominoes

- Need to give base cases:
  - $T_2 = 3,\ T_1 = 0$
  - $A_1 = 1$
  - $B_1 = 1$

Example 2
Dominoes

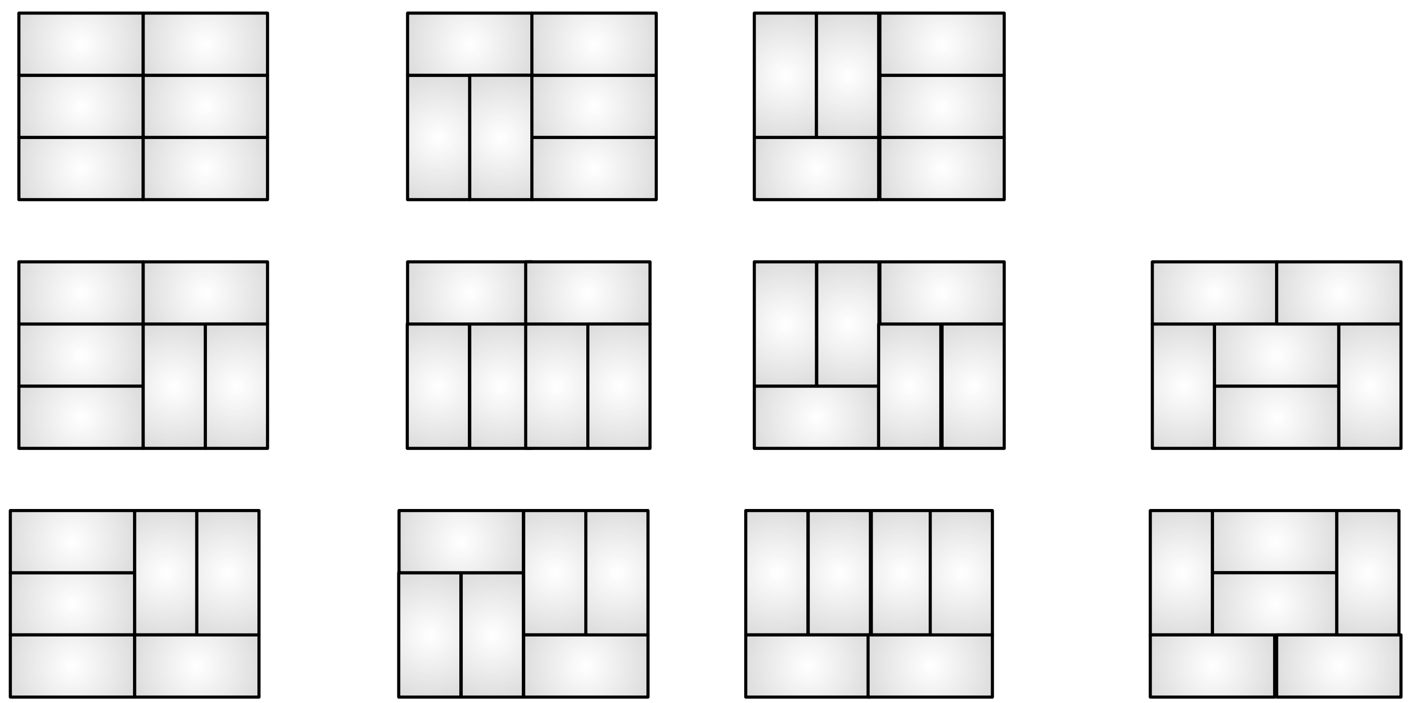
- Now we can calculate:
  - $A_2 = A_0 + T_1 = 0 + 0 = 0$
  - $B_2 = B_0 + T_1 = 0 + 0 = 0$

  - $T_3 = T_1 + A_2 + B_2 = 0 + 0 + 0$
  - $A_3 = A_1 + T_2 = 1 + 3 = 4$
  - $B_3 = B_1 + T_2 = 1 + 3 = 4$



Example 2
Dominoes

- $T_4 = T_2 + A_3 + B_3 = 3 + 4 + 4 = 11$



- $A_4 = T_3 + A_2 = 0 + 0 = 0$
- $B_4 = T_3 + B_2 = 0 + 0 = 0$

Example 2
Dominoes

- $T_5 = T_3 + A_4 + B_4 = 0 + 0 + 0 = 0$
- $A_5 = T_4 + A_3 = 11 + 4 = 15$
- $B_5 = T_4 + B_3 = 11 + 4 = 15$

Example 2
Dominos

- $T_6 = T_4 + A_5 + B_5 = 11 + 15 + 15 = 41$

- If you have a domino set (or two) try out to generate them all

- If we continue, we notice that if the number of columns is odd, there is **no** tiling.

Implementing Dominos

- Method 1: Tabulation
  - Create three arrays for A, B, and T
  - Seed them with the initial values
  - Then calculate

n	T	A	B
0	0	0	0
1	0	1	1
2	3	0	0
3			
4			
5			
6			
7			
8			
9			

Implementing Dominos

- Method 1: Tabulation
  - Create three arrays for A, B, and T
  - Seed them with the initial values
  - Then calculate
    - T[3] = T[1]+A[2]+B[2]
    - The needed values are already in the table

n	T	A	B
0	0	0	0
1	0	1	1
2	3	0	0
3			
4			
5			
6			
7			
8			
9			

Implementing Dominos

- Method 1: Tabulation
  - Create three arrays for A, B, and T
  - Seed them with the initial values
  - Then calculate
    - A[3] =A[1]+T[2]
    - The needed values are already in the table

n	T	A	B
0	0	0	0
1	0	1	1
2	3	0	0
3	0	**4**	
4			
5			
6			
7			
8			
9			

Implementing Dominos

- Method 1: Tabulation
  - Create three arrays for A, B, and T
  - Seed them with the initial values
  - Then calculate
    - A[3] =A[1]+T[2]
    - The needed values are already in the table

n	T	A	B
0	0	0	0
1	0	1	1
2	3	0	0
3	0	4	4
4	11	0	0
5	0	15	15
6	41	0	0
7	0	56	56
8	153	0	0
9	0	209	209

Implementing Dominoes

- We can do this in Python as well
    - In order not to get confused with indexing, I generate arrays that are filled with zeroes but for the initial values

```
def dominoes(n):
    A = [0, 1, 0]
    B = [0, 1, 0]
    T = [0, 0, 3]
    for _ in range(n-2):
        A.append(0)
        B.append(0)
        T.append(0)
    for i in range(3,n+1):
        T[i] = T[i-2]+A[i-1]+B[i-1]
        A[i] = T[i-1]+A[i-2]
        B[i] = T[i-1]+B[i-2]
    return T, A, B
```

Implementing Dominoes

- This can be improved.
  - Filling with zeroes is for zombies
  - A and B array of course have the same contents, so we only need one
- However, an exploding function like this one is deeply satisfying.
  - There are 742458041222319620289594938481080302679457061763190907611166881866538519953377270264063604929093882393253033609205329571467793832497625768032973504282592233225474265618533768828288692682770245729919519477949798330567599374579693741711474060137712819536523442291124744696316634042777721 tessellations of a 1000 by 3 field

Implementing Dominoes

- Method 2: Memoization (sic)
  - We really have a system of recurrences here
    - $T(n) = T(n-2) + A(n-1) + B(n-1)$
      $= T(n-2) + 2A(n-1)$
      - after getting rid of the A-copy B
    - $A(n) = A(n-2) + T(n-1)$
  - So, we could use recursion
    - This will actually work, because we never calculate the same value twice.

Implementing Dominoes

```
def t(n):
    if n < 2:
        return 0
    if n == 2:
        return 3
    else:
        return t(n-2)+2*a(n-1)

def a(n):
    if n<1:
        return 0
    if n==1:
        return 1
    else:
        return a(n-2)+t(n-1)
```

Implementing Dominoes

- But remember recursive Fibonacci?
  - We could spend a lot of time recalculating values
- This is where memoization comes in.
  - Remember all of the previous values in a dictionary
  - In our case, we need one dictionary for T values and another for A values

Implementing Dominoes

- Let's speed up Fibonacci number calculation

```
def recfib(n):
    if n < 2:
        return n
    else:
        return recfib(n-1)+recfib(n-2)
```

Implementing Dominoes

- We want to remember the recursive results
- Create a dictionary with partially filled in values
- When we calculate recursively
    - See whether a requested value is already in the dictionary
    - Add any calculated value to the dictionary

Implementing Dominoes

- Defining the dictionary
  - Step 1: Declare the dictionary
    - Dictionary needs to stay the same between different calls to recfib, so it cannot be a local variable
    - We can but do not need to use global
      - Because dictionaries are called by reference, we can change them from within a function

Implementing Dominoes

```
fibdic={0:0, 1:1}

def memfib(n):
    global fibdic
    if n in fibdic:
        return fibdic[n]
    else:
        value = memfib(n-1)+memfib(n-2)
        fibdic[n]=value
        return value
```

Dictionary is defined outside the function's body

Implementing Dominoes

```
fibdic={0:0, 1:1}

def memfib(n):
    global fibdic
    if n in fibdic:
        return fibdic[n]
    else:
        value = memfib(n-1)+memfib(n-2)
        fibdic[n]=value
        return value
```

The base case is now encoded in the dictionary

Implementing Dominoes

```
fibdic={0:0, 1:1}

def memfib(n):
    global fibdic
    if n in fibdic:
        return fibdic[n]
    else:
        value = memfib(n-1)+memfib(n-2)
        fibdic[n]=value
        return value
```

This is not really necessary
Scope rules would find fibdic

Implementing Dominoes

```
fibdic={0:0, 1:1}

def memfib(n):
    global fibdic
    if n in fibdic:
        return fibdic[n]
    else:
        value = memfib(n-1)+memfib(n-2)
        fibdic[n]=value
        return value
```

First look for the value in the dictionary.

Implementing Dominoes

```
fibdic={0:0, 1:1}

def memfib(n):
    global fibdic
    if n in fibdic:
        return fibdic[n]
    else:
        value = memfib(n-1)+memfib(n-2)
        fibdic[n]=value
        return value
```

When we do a calculation, we need to do two things:
(1) Update the dictionary
(2) Return the result

Implementing Dominoes

```
fibdic={0:0, 1:1}

def memfib(n):
    global fibdic
    if n in fibdic:
        return fibdic[n]
    else:
        value = memfib(n-1)+memfib(n-2)
        fibdic[n]=value
        return value
```

(1) Update the dictionary

Implementing Dominoes

```
fibdic={0:0, 1:1}

def memfib(n):
    global fibdic
    if n in fibdic:
        return fibdic[n]
    else:
        value = memfib(n-1)+memfib(n-2)
        fibdic[n]=value
        return value
```

(2) Return the result

Implementing Dominoes

- This now runs much faster
    - And the dictionary persists between calls

```
>>> memfib(1000)
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492288 75
>>> fibdic
Squeezed text (1396 lines).
>>>
Ln: 134    Col: 0
```

Implementing Memoization

- In Python, define a decorator
  - A decorator applies a function on a function to be defined
  - When the function is called, the function of the function is instead called
  - Same function can apply to many different functions

Implementing Memoization

```
def memoize_function(f):
    memory = {}
    def inner(num):
        if num not in memory:
            memory[num] = f(num)
        return memory[num]
    return inner
```

The outer function

Implementing Memoization

```
def memoize_function(f):
    memory = {}
    def inner(num):
        if num not in memory:
            memory[num] = f(num)
        return memory[num]
    return inner
```

memory is available inside the full scope of memoize_function

Implementing Memoization

```
def memoize_function(f):
    memory = {}
    def inner(num):
        if num not in memory:
            memory[num] = f(num)
        return memory[num]
    return inner
```

inner is the function that we return

Implementing Memoization

```
def memoize_function(f):
    memory = {}
    def inner(num):
        if num not in memory:
            memory[num] = f(num)
        return memory[num]
    return inner
```

inner is the function that we return

Implementing Memoization

```
def memoize_function(f):
    memory = {}
    def inner(num):
        if num not in memory:
            memory[num] = f(num)
        return memory[num]
    return inner
```

inner is the function that we return

Implementing Memoization

```
@memoize_function
def fib(n):
    if n < 0:
        return 0
    if n <= 1:
        return n
    else:
        return fib(n-1)+fib(n-2)
```

This is a decorator
Just write @ and the name of the function of a function

Implementing Memoization

```
@memoize_function
def fib(n):
    if n < 0:
        return 0
    if n <= 1:
        return n
    else:
        return fib(n-1)+fib(n-2)
```

This is the normal recursive function

Without the decorator, it would run very slowly, but now it is very fast.

Implementing Memoization

```
import functools

@functools.cache
def fib(n):
    if n < 0:
        return 0
    if n <= 1:
        return n
    else:
        return fib(n-1)+fib(n-2)
```

Using the module functools, we do not even have to implement it ourselves

Dynamic Programming

- Three steps:
  - Define sub-problems
  - Set-up a recursion
  - Determine base cases

Knapsack Problem

- Continuous knapsack problem
  - Select items from set $X = \{A_1, A_2, \ldots, A_n\}$
  - Each item has a weight w_i
  - Each item has a value v_i
  - Maximize $\sum_{i \in M} s_i v_i$ subject to $\sum_{i \in M} s_i w_i \leq C$
    - with $s_i \in [0,1]$

Continuous Knapsack Problem

- Story:
  - You have sent a mining robot to an asteroid.
  - Mining asteroids has been suspended by the world government and on April 1st, you need to abandon all activity
  - You have one last freight to send to earth with a capacity of 10 tons.
    - You mining operation has yielded:
      - 2 tons paladium ($2600 per ounce)
      - 7 tons platinum ($750 per ounce)
      - 3 tons gold ($1700 per ounce)
      - 4 tons silver ($15 per ounce)
  - What do you select for your final journey

Continuous Knapsack Problem

- Solution:
  - You load the rocket with what you have in order of preciousness:
    - All the Paladium, all the gold, and what you can of the platinum

Knapsack Problem

- Continuous knapsack problem
  - Greedy algorithm solves the continuous knapsack algorithm:
    - Order items by ratios of value over weight
      - Select items in order of this ratio
        - As long as remaining under capacity
      - Last item might be fractional

Knapsack Problem

- Example

Item	Value	Weight	Ratio
A	9	5	1.80
B	7	4	1.75
C	6	4	1.5
D	3	2	1.5
E	2	2	1
F	1	1	1

- Total capacity is 6

Knapsack Problem

- Example

Item	Value	Weight	Ratio
A	9	5	1.80
B	7	4	1.75
C	6	4	1.5
D	3	2	1.5
E	2	2	1
F	1	1	1

- $s_A = 1,\ s_B = 0.25,\ s_C = s_D = s_E = s_F = 0$
- Total capacity is 6, total value is 10.75

Knapsack Problem

- 0-1 knapsack
    - Can select only an entire item, but not a fraction
- Greedy method is no longer best

Knapsack Problem

- 0-1 knapsack
  - Story:
    - You are the director of the Louvre
    - You are told that there is an asteroid crashing into the museum in 5 hours
    - You estimate that you can move 10 tons of art to a safe place in the time you have left
    - Which pieces do you select?
  - If you start with the most conspicuous items, you select the Nike of Samothrace, Mona Lisa, and your Rubens collection
    - But if you give up on the multi-ton Nike of Samothrace, you can safe almost all of the famous paintings you have

Knapsack Problem

- Example

Item	Value	Weight	Ratio
A	9	5	1.80
B	7	4	1.75
C	6	4	1.5
D	3	2	1.5
E	2	2	1

- Total capacity is 6

Knapsack Problem

- Example

Item	Value	Weight	Ratio
A	9	5	1.80
B	7	4	1.75
C	6	4	1.5
D	3	2	1.5
E	2	2	1

- Greedy solution: $s_A = 1,\ s_B = s_C = s_D = s_E = s_F = 0$
  - Select only A
- Total weight is 5 and total gain is 9

Knapsack Problem

- Example

Item	Value	Weight	Ratio
A	9	5	1.80
B	7	4	1.75
C	6	4	1.5
D	3	2	1.5
E	2	2	1
F	1	1	1

- Better solution: $s_B = 1,\ s_D = 1, s_A = s_C = s_E = s_F = 0$
  - Include B and D
- Total weight is 6 and total value is 10

Knapsack Problem

- One possibility:
  - Enumerate and evaluate all possible combinations of items
  - This means checking 2^n combinations of items
    - One for each subset.
- Solving knapsack problems with dynamic programming
  - Sub-problems?
  - Recursion?
  - Base Case?

Knapsack Problem

- Sub-problems
  - Optimal solution needs to be composed of solutions for subproblems
  - Use less items, use fewer capacities

Knapsack Problem

- Order all items in any order
- Optimal solution:
  - Two alternatives:
    - Last item is included
    - Last item is not included

Knapsack Problem

- Order all items in any order
  - Optimal solution:
    - Two alternatives:
      - Last item is included
        - Before inclusion of the last item
          - Solved knapsack for all but last item with total capacity minus weight of last item
      - Last item is not included
        - Before non-inclusion of the last item
          - Solved knapsack for all but last item with total capacity

Knapsack Problem

- Recursion:

$$g(w, X_0, X_1, \ldots X_n) = \max(g(w, X_0, X_1, \ldots, X_{n-1}), g(w - w(X_n), X_0, X_1, \ldots X_{n-1})$$

Knapsack Problem

- Generate Table
  - Columns: set of items is

    $$\{A_0\},\ \{A_0, A_1\},\ \{A_0, A_1, A_2\}, \{A_0, A_1, A_2, A_3\}, \ldots$$

  - Rows: Capacity below problem capacity

Knapsack Problem

- Example

Item	Value	Weight	Ratio
A	9	5	1.80
B	7	4	1.75
C	6	4	1.5
D	3	2	1.5
E	2	2	1
F	1	1	1

- Total capacity is 6

Knapsack Problem

- Example

Item	Value	Weight	Ratio
A	9	5	1.80
B	7	4	1.75
C	6	4	1.5
D	3	2	1.5
E	2	2	1
F	1	1	1

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						
4	0						
5	0						
6	0						

- Cell in column {*A*, … , *X*} and row r is the gain of selecting from {*A*, … , *X*} and maximum capacity *r*

Knapsack Problem

- Element in row *r* and columns X_i

$$g_{r,X_i} = \begin{cases} g_{r,X_{i-1}} & \textbf{if } X_i \textbf{ is not selected} \\ g_{r-w_i,X_{i-1}} + v_i & \textbf{if } X_i \textbf{ is selected} \end{cases}$$

$$= \max\left(g_{r,X_{i-1}}, g_{r-w_i,X_{i-1}} + v_i\right)$$

Knapsack Problem

- Base cases:
  - No items to select: gain is zero
  - Capacity is zero: gain is zero

Knapsack Problem

- Work **forward** adding column after column
  - Item A has weight 5 and value 9

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0					
2	0	0					
3	0	0					
4	0	0					
5	0	9					
6	0	9					

Knapsack Problem

- Item B has weight 4 and value 7

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0				
2	0	0	0				
3	0	0	0				
4	0	0	7				
5	0	9	max(7,9)				
6	0	9	max(7,9)				

Knapsack Problem

- Item C has value 6 and weight 4

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0			
2	0	0	0	0			
3	0	0	0	0			
4	0	0	7	max(6,7)			
5	0	9	9	max(6, 9)			
6	0	9	9	max(6, 9)			

Knapsack Problem

- Item *D* has weight 2 and value 3

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0		
2	0	0	0	0	3		
3	0	0	0	0	3		
4	0	0	7	7	max(7,3)		
5	0	9	9	9	max(9,3)		
6	0	9	9	9	max(9,10)		

Knapsack Problem

- Item *E* has weight 2 and value 2

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	
2	0	0	0	0	3	max(3,2)	
3	0	0	0	0	3	max(3,2)	
4	0	0	7	7	7	max(7,5)	
5	0	9	9	9	9	max(9,5)	
6	0	9	9	9	10	max(10,9)	

Knapsack Problem

- Item F has weight 1 and value 1

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	max(1,0)
2	0	0	0	0	3	3	max(3,1)
3	0	0	0	0	3	3	max(3,4)
4	0	0	7	7	7	7	max(7,4)
5	0	9	9	9	9	9	max(9,8)
6	0	9	9	9	10	10	max(10,10)

Knapsack Problem

- Final table tells us the realizable total value

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10

  - but not how to obtain it

Knapsack Problem

- Can either annotate table entry with how we got them

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10

- or can backtrack

Knapsack Problem

- Backtracking

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10=9+1

- Last entry is either with or without including F

Knapsack Problem

- Backtracking

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10=9+1

- Let’s say we include it

Knapsack Problem

- Backtracking
  - Then the 10 was realized as 9+1 with the previous column and row - weight of item F =1

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10=9+1

Knapsack Problem

- Backtracking
  - No such choice with the other ones until we get to *A*

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10=9+1

Knapsack Problem

- Backtracking

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10=9+1

- Included A and F for a total value of 10 and a total weight of 6

Knapsack Problem

- Backtracking alternative in the first step:

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10

- Don’t include F, E

Knapsack Problem

- Backtracking

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10

  - must have included item D

Knapsack Problem

- Backtracking

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10

- D has value 3 and weight 2

Knapsack Problem

- Backtracking

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10

- Do not include C

Knapsack Problem

- Backtracking

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10

- But include B

Knapsack Problem

- Backtracking

	{ }	{A}	{A,B}	{A,B,C}	{A,B,C,D}	{A, …, E}	{A, …, F}
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	3	3	3
3	0	0	0	0	3	3	4
4	0	0	7	7	7	7	7
5	0	9	9	9	9	9	9
6	0	9	9	9	10	10	10

  - and therefore not A

- Alternative solution: Select B and D for the same total value and capacity

Knapsack Problem

- Your turn: Extend to total capacity 10

Item	Value	Weight
A	9	5
B	7	4
C	6	4
D	3	2
E	2	2
F	1	1

Cap	A	B	C	D	E	F
1	0	0	0	0	0	1
2	0	0	0	3	3	3
3	0	0	0	3	3	4
4	0	7	7	7	7	7
5	9	9	9	9	9	9
6	9	9	9	10	10	10
7	9	9	9	12	12	12
8	9	9	13	13	13	13
9	9	16	16	16	16	16
10	9	16	16	16	16	17

Knapsack Problem

Cap	A	B	C	D	E	F
1	0	0	0	0	0	1
2	0	0	0	3	3	3
3	0	0	0	3	3	4
4	0	7	7	7	7	7
5	9	9	9	9	9	9
6	9	9	9	10	10	10
7	9	9	9	12	12	12
8	9	9	13	13	13	13
9	9	16	16	16	16	16
10	9	16	16	16	16	17

Include F

Knapsack Problem

Cap	A	B	C	D	E	F
1	0	0	0	0	0	1
2	0	0	0	3	3	3
3	0	0	0	3	3	4
4	0	7	7	7	7	7
5	9	9	9	9	9	9
6	9	9	9	10	10	10
7	9	9	9	12	12	12
8	9	9	13	13	13	13
9	9	16	16	16	16	16
10	9	16	16	16	16	17

```
Include F
Do not
include E
```

Knapsack Problem

Cap	A	B	C	D	E	F
1	0	0	0	0	0	1
2	0	0	0	3	3	3
3	0	0	0	3	3	4
4	0	7	7	7	7	7
5	9	9	9	9	9	9
6	9	9	9	10	10	10
7	9	9	9	12	12	12
8	9	9	13	13	13	13
9	9	16	16	16	16	16
10	9	16	16	16	16	17

```
Include F
Do not
include E
Do not
include D
```

Knapsack Problem

Cap	A	B	C	D	E	F
1	0	0	0	0	0	1
2	0	0	0	3	3	3
3	0	0	0	3	3	4
4	0	7	7	7	7	7
5	9	9	9	9	9	9
6	9	9	9	10	10	10
7	9	9	9	12	12	12
8	9	9	13	13	13	13
9	9	16	16	16	16	16
10	9	16	16	16	16	17

```
Include F
Do not
include E
Do not
include D
Do not
include C
```

Knapsack Problem

Cap	A	B	C	D	E	F
1	0	0	0	0	0	1
2	0	0	0	3	3	3
3	0	0	0	3	3	4
4	0	7	7	7	7	7
5	9	9	9	9	9	9
6	9	9	9	10	10	10
7	9	9	9	12	12	12
8	9	9	13	13	13	13
9	9	16	16	16	16	16
10	9	16	16	16	16	17

```
Include F
Do not
include E
Do not
include D
Do not
include C
Include B
```

Knapsack Problem

Cap	A	B	C	D	E	F
1	0	0	0	0	0	1
2	0	0	0	3	3	3
3	0	0	0	3	3	4
4	0	7	7	7	7	7
5	9	9	9	9	9	9
6	9	9	9	10	10	10
7	9	9	9	12	12	12
8	9	9	13	13	13	13
9	9	16	16	16	16	16
10	9	16	16	16	16	17

```
Include F
Do not
include E
Do not
include D
Do not
include C
Include B
Include A
```

0-1 Knapsack Example

- An alternative to backtracking is to store the information in the table

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

0-1 Knapsack Example

- Create the table and initialize it

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G
0	0							
1	0							
2	0							
3	0							
4	0							
5	0							
6	0							
7	0							
8	0							
9	0							
10	0							

0-1 Knapsack Example

- The column for A is simple.
- Red means: item included

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

```
Total Capacity 10
```

	/	A	B	C	D	E	F	G
0	0	0						
1	0	0						
2	0	0						
3	0	0						
4	0	0						
5	0	10						
6	0	10						
7	0	10						
8	0	10						
9	0	10						
10	0	10						

0-1 Knapsack Example

- The column for A is simple.
- Red means: item included

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G
0	0	0	0					
1	0	0	0					
2	0	0	0					
3	0	0	0					
4	0	0	9					
5	0	10	10	max(10,9)				
6	0	10	10					
7	0	10	10					
8	0	10	10					
9	0	10	19					
10	0	10	19					

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G
0	0	0	0	0				
1	0	0	0	0				
2	0	0	0	0				
3	0	0	0	0				
4	0	0	9	9				
5	0	10	10	10				
6	0	10	10					
7	0	10	10					
8	0	10	10					
9	0	10	19					
10	0	10	19					

10 = max(10, 0+8)

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G
0	0	0	0	0				
1	0	0	0	0				
2	0	0	0	0				
3	0	0	0	0				
4	0	0	9	9				
5	0	10	10	10				
6	0	10	10	10				
7	0	10	10	10				
8	0	10	10	17				
9	0	10	19	19				
10	0	10	19	19				

`17 = max(10, 9 + 8)`

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G
0	0	0	0	0	0			
1	0	0	0	0	0			
2	0	0	0	0	0			
3	0	0	0	0	0			
4	0	0	9	9	9			
5	0	10	10	10	10			
6	0	10	10	10	10			
7	0	10	10	10	10			
8	0	10	10	17	17			
9	0	10	19	1				
10	0	10	19	19				

`17 = max(17, 9 + 7)`

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0		
1	0	0	0	0	0	0		
2	0	0	0	0	0	0		
3	0	0	0	0	0	7		
4	0	0	9	9	9			
5	0	10	10	10	10			
6	0	10	10	10	10			
7	0	10	10	10	10			
8	0	10	10	17	17			
9	0	10	19	19	19			
10	0	10	19	19	19			

`7 = max(0, 7)`

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0		
1	0	0	0	0	0	0		
2	0	0	0	0	0	0		
3	0	0	0	0	0	7		
4	0	0	9	9	9	9		
5	0	10	10	10				
6	0	10	10	10	10			
7	0	10	10	10	10			
8	0	10	10	17	17			
9	0	10	19	19	19			
10	0	10	19	19	19			

9 = max(9, 7)

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0		
1	0	0	0	0	0	0		
2	0	0	0	0	0	0		
3	0	0	0	0	0	7		
4	0	0	9	9	9	9		
5	0	10	10	10	10	10		
6	0	10	10	10	10	10		
7	0	10	10	10	10	16		
8	0	10	10	17	17	17		
9	0	10	19	19	19			
10	0							

This is a tie!
Break it as you wish

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0		
1	0	0	0	0	0	0		
2	0	0	0	0	0	0		
3	0	0	0	0	0	7		
4	0	0	9	9	9	9		
5	0	10	10	10	10	10		
6	0	10	10	10	10	10		
7	0	10	10	10	10	16		
8	0	10	10	17	17	17		
9	0	10	19	19	19	19		
10	0	10	19	19	19	19		

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	
3	0	0	0	0	0	7	7	
4	0	0	9	9	9	9	9	
5	0	10	10	10	10	`max(9, 0+4)`		
6	0	10	10	10	10	10		
7	0	10	10	10	10	16		
8	0	10	10	17	17	17		
9	0	10	19	19	19	19		
10	0	10	19	19	19	19		

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	
3	0	0	0	0	0	7	7	
4	0	0	9	9	9	9	9	
5	0	10	10	10	10	10	11	
6	0	10	10	10	10	`max(10, 7+4)`		
7	0	10	10	10	10	16		
8	0	10	10	17	17	17		
9	0	10	19	19	19	19		
10	0	10	19	19	19	19		

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

```
Total Capacity 10
```

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	
3	0	0	0	0	0	7	7	
4	0	0	9	9	9	9	9	
5	0	10	10	10	10	10	11	
6	0	10	10	10	10	10	13	
7	0	10	10	10	1[illegible]			
8	0	10	10	17	17	17		
9	0	10	19	19	19	19		
10	0	10	19	19	19	19		

```
max(10, 9+4)
```

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	
3	0	0	0	0	0	7	7	
4	0	0	9	9	9	9	9	
5	0	10	10	10	10	10	11	
6	0	10	10	10	10	10	13	
7	0	10	10	10	10	16	16	
8	0	10	10	17	[illegible]			
9	0	10	19	19	19	19		
10	0	10	19	19	19	19		

max(16, 10+4)

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	
3	0	0	0	0	0	7	7	
4	0	0	9	9	9	9	9	
5	0	10	10	10	10	10	11	
6	0	10	10	10	10	10	13	
7	0	10	10	10	10	16	16	
8	0	10	10	17	17	17	17	
9	0	10	19	19	1			
10	0	10	19	19	19	19		

`max(17, 10+4)`

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	
3	0	0	0	0	0	7	7	
4	0	0	9	9	9	9	9	
5	0	10	10	10	10	10	11	
6	0	10	10	10	10	10	13	
7	0	10	10	10	10	16	16	
8	0	10	10	17	17	17	17	
9	0	10	19	19	19	19	20	
10	0	10	19	1				

`max(19, 16+4)`

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

```
Total Capacity 10
```

	/	A	B	C	D	E	F	G
0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	
3	0	0	0	0	0	7	7	
4	0	0	9	9	9	9	9	
5	0	10	10	10	10	10	11	
6	0	10	10	10	10	10	13	
7	0	10	10	10	10	16	16	
8	0	10	10	17	17	17	17	
9	0	10	19	19	19	19	20	
10	0	10	19	19	19	19	21	

```
max(19, 17+4)
```

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	[illegible]	[illegible]	[illegible]		
6	0	10	10	10	[illegible]	[illegible]	[illegible]		
7	0	10	10	10	10	16	16		
8	0	10	10	17	17	17	17		
9	0	10	19	19	19	19	20		
10	0	10	19	19	19	19	21		

`max(9, 4+3)`

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	[illegible]	[illegible]	[illegible]		
7	0	10	10	10	[illegible]	[illegible]	[illegible]		
8	0	10	10	17	17	17	17		
9	0	10	19	19	19	19	20		
10	0	10	19	19	19	19	21		

max(11, 7+3)

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16		
8	0	10	10	17	max(13, 9+3)				
9	0	10	19	19	19	19	20		
10	0	10	19	19	19	19	21		

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17		
9	0	10	19	19					
10	0	10	19	19	19	19	21		

`max(16, 11+3)`

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19					
10	0	10	19	19	19	19	21		

max(17, 13+3)

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19						

max(20, 16+3)

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	

max(21, 17+3)

0-1 Knapsack Example

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

`max(21, 17+1)`

0-1 Knapsack Example

- Backtracking is now easier

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do not include H

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do not include G

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do include F

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do include F
  - F has weight 2

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do include F
  - F has weight 2

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do not include E

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do not include D

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do include C

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do include C
  - C has weight 4

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do include B

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

`Total Capacity 10`

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- Do include B
  - B has weight 4

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

0-1 Knapsack Example

- We have reached the upper row
-

Item	Value	Weight
A	10	5
B	9	4
C	8	4
D	7	4
E	7	3
F	4	2
G	3	2
H	1	2

Total Capacity 10

	/	A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	4	4	
3	0	0	0	0	0	7	7	7	
4	0	0	9	9	9	9	9	9	
5	0	10	10	10	10	10	11	11	
6	0	10	10	10	10	10	13	13	
7	0	10	10	10	10	16	16	16	
8	0	10	10	17	17	17	17	17	
9	0	10	19	19	19	19	20	20	
10	0	10	19	19	19	19	21	21	21

Knapsack Problem

- Multiple Item Selection
  - When considering item *j*, need to look at including ν items $\nu \in \{0,1,\ldots,\lfloor \frac{i}{w_j} \rfloor\}$
  - Formula changes
    - $g_{i,j} = \max(\{g_{i-\nu w_j} + \nu v_j \mid \nu \in \{0,1,\ldots,\lfloor \frac{i}{w_j} \rfloor\})$

Knapsack Problem

- Example: Items with value-weight of (26,5), (20,4), (14,3), (9,2), (4,1)

Knapsack Problem

(26,5), (20,4), (14,3), (9,2), (4,1)

TW	A	B	C	D	E
0	0	0	0	0	0
1	0	0	0	0	4
2	0	0	0	9	9
3	0	0	14	14	14
4	0	20	20	20	20
5	26	26	26	26	26
6	26	26	28	29	30
7	26	26	34	35	35
8	26	40	40	40	40
9	26	46	46	46	46
10	52	52	52	52	52
11	52	52	54	55	56
12	52	60	60	61	61
13	52	66	66	66	66
14	52	72	72	72	72
15	78	78	78	78	78
16	78	80	80	81	82

Knapsack Problem

(26,5), (20,4), (14,3), (9,2), (4,1)

TW	A	B	C	D	E
0	0	0	0	0	0
1	0	0	0	0	4
2	0	0	0	9	9
3	0	0	14	14	14
4	0	20	20	20	20
5	26	26	26	26	26
6	26	26	28	29	30
7	26	26	34	35	35
8	26	40	40	40	40
9	26	46	46	46	46
10	52	52	52	52	52
11	52	52	54	55	56
12	52	60	60	61	61
13	52	66	66	66	66
14	52	72	72	72	72
15	78	78	78	78	78
16	78	80	80	81	82

Backtrack to find optimal selection

Knapsack Problem

`(26,5), (20,4), (14,3), (9,2), (4,1)`

TW	A	B	C	D	E
0	0	0	0	0	0
1	0	0	0	0	4
2	0	0	0	9	9
3	0	0	14	14	14
4	0	20	20	20	20
5	26	26	26	26	26
6	26	26	28	29	30
7	26	26	34	35	35
8	26	40	40	40	40
9	26	46	46	46	46
10	52	52	52	52	52
11	52	52	54	55	56
12	52	60	60	61	61
13	52	66	66	66	66
14	52	72	72	72	72
15	78	78	78	78	78
16	78	80	80	81	82

```
Optimal solution:

3 items of type A
1 item of type E
```

Matrix Chain Multiplication

Matrix Chain Multiplication

- Given *n* integer matrices of various dimensions

$$A_1, A_2, A_3, \ldots, A_n$$

- Task is: multiply the matrices with the least number of multiplications of coefficients

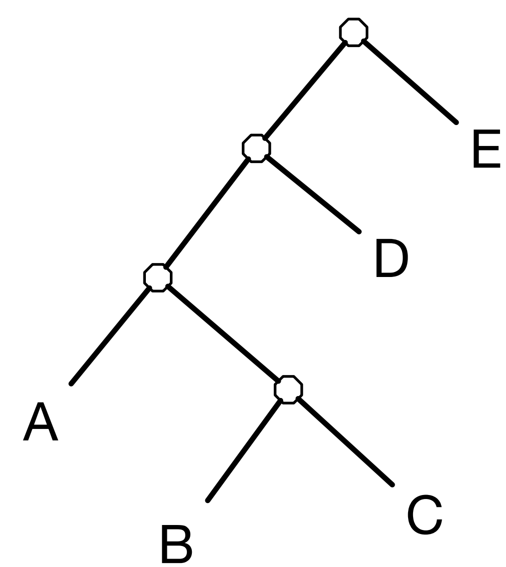
$$A_1 \times A_2 \times A_3 \times \ldots \times A_n$$

  - We can change the order in which we execute the multiplications

Matrix Chain Multiplication

- Different parenthesization have different costs
  - Parenthesization corresponds to different evaluation trees

$((A(BC))D)E$



Matrix Chain Multiplication

- Dynamic programming approach
  - A product of *n* matrices is made up of one of the following
    - Product of 1 with product of *n*-1 matrices
    - Product of 2 with product of *n-2* matrices
    - …
    - Product of *n*-2 with product of 2 matrices
    - Product of *n*-1 with product of 1 matrix

Matrix Chain Multiplication

- Example:
  - A 5×7
  - B 7×2
  - C 2×10
  - D 10×4
  - E 4×5

Matrix Chain Multiplication

- Start with product of two matrices in order
- $AB \quad 5 \times 7 \times 2 = 70$

$BC \quad 7 \times 2 \times 10 = 140$

$CD \quad 2 \times 10 \times 4 = 80$

$DE \quad 10 \times 4 \times 5 = 200$

$A : 5 \times 7; \quad B : 7 \times 2; \quad C : 2 \times 10; \quad D : 10 \times 4; \quad E : 4 \times 5$

Matrix Chain Multiplication

- Then products of three

$A(BC) \quad 5 \times 7 \times 10 + 140 = 490 \qquad (AB)C \quad 5 \times 2 \times 10 + 70 = 170$

$B(CD) \quad 7 \times 2 \times 4 + 80 = 136 \qquad (BC)D \quad 7 \times 10 \times 4 + 140 = 420$

$C(DE) \quad 2 \times 10 \times 5 + 200 = 300 \qquad (CD)E \quad 80 + 2 \times 4 \times 5 = 120$

$A : 5 \times 7; \quad B : 7 \times 2; \quad C : 2 \times 10; \quad D : 10 \times 4; \quad E : 4 \times 5$

Matrix Chain Multiplication

- And products of four $ABCD$

$$(AB)(CD) \quad 5 \times 2 \times 4 + 70 + 80 = 190$$
$$A(BCD) \quad 5 \times 7 \times 4 + 136 = 276$$
$$(ABC)D \quad 170 + 5 \times 10 \times 4 = 370$$

- $BCDE$

$$(BC)(DE) \quad 7 \times 10 \times 5 + 140 + 200 = 690$$
$$B(CDE) \quad 7 \times 2 \times 5 + 120 = 190$$
$$(BCD)E \quad 7 \times 4 \times 5 + 136 = 276$$

$$A : 5 \times 7; \quad B : 7 \times 2; \quad C : 2 \times 10; \quad D : 10 \times 4; \quad E : 4 \times 5$$

Matrix Chain Multiplication

- And finally the complete product

$$A(BCDE): \quad 5 \times 7 \times 5 + 190 = 175 + 190 = 285$$

$$(AB)(CDE): \quad 5 \times 2 \times 5 + 70 + 120 = 50 + 190 = 240$$

$$(ABC)(DE): \quad 5 \times 10 \times 5 + 170 + 200 = 250 + 370 = 620$$

$$(ABCD)E: \quad 5 \times 4 \times 5 + 190 = 100 + 190 = 290$$

$$A: 5 \times 7; \quad B: 7 \times 2; \quad C: 2 \times 10; \quad D: 10 \times 4; \quad E: 4 \times 5$$

Matrix Chain Multiplication

- How to best organize the calculation?

A	B	C	D	E
0	0	0	0	0

AB	BC	CD	DE
70	140	80	200

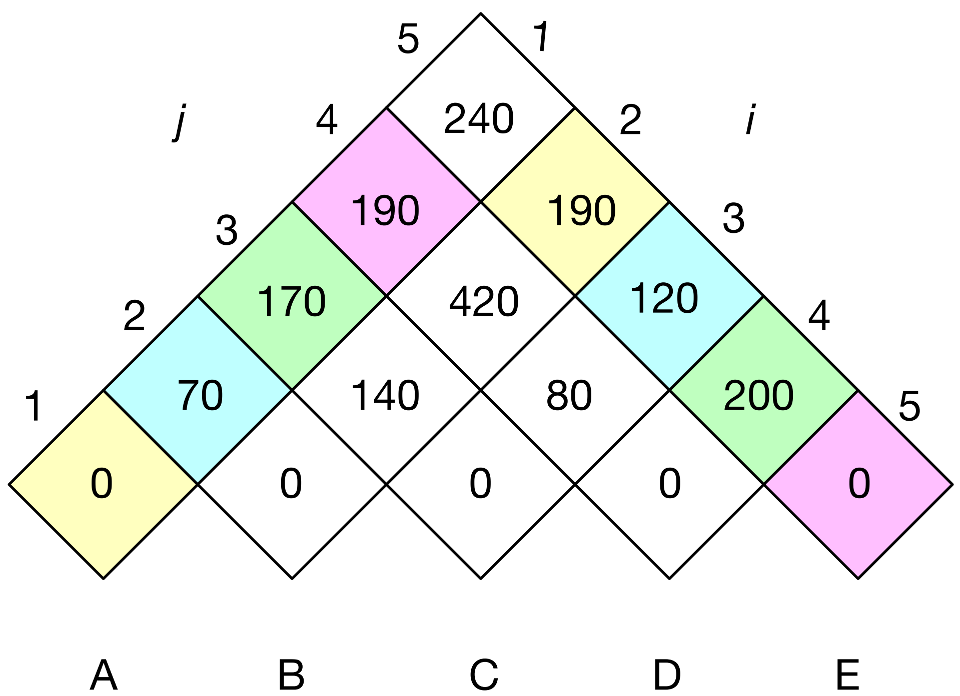
ABC	BCD	CDE
170	136	120

ABCD	BCDE
190	190

ABCDE
240

Matrix Chain Multiplication

- Another way to look at it:



$$240 = \min(190 + \textbf{costs}(A, BCDE), 120 + 70 + \textbf{costs}(AB, CDE), 200 + 170 + \textbf{costs}(ABC, DE), 0 + 190 + \textbf{costs}(ABCD, E))$$

Matrix Chain Multiplication

- Arrange the sizes of the matrices in an array sizes
- Matrix A_i has size `sizes[i-1] x sizes[i]`
- Recursively, define
    - `m[i][j] = 0` if `i==j`
    - `m[i][j] = min([ m[i][k]+m[k+1][j] + sizes[i-1]sizes[k]sizes[j] for k in range(i,j)])`
- To remember our choice for *k*, we mark it in an array
    - `best[i][j] = k`

Matrix Chain Multiplication

- Implementation:
  - We can either fill in the two arrays (m and best)
  - Or we can use memoization with the recursion

Levenshtein Distance

Levenshtein Distance

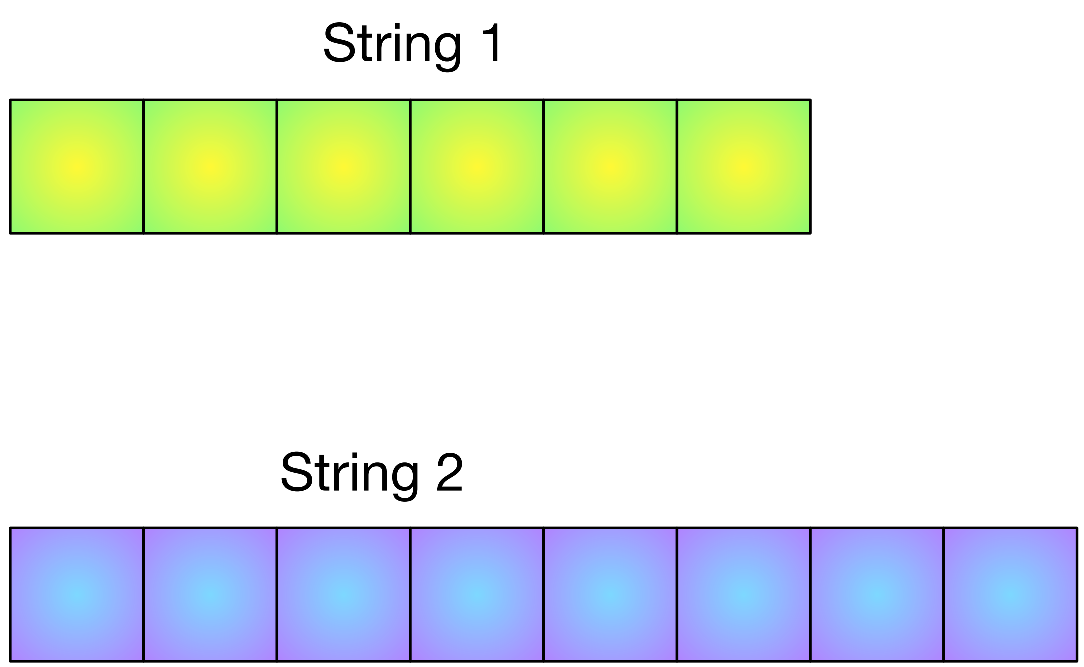
- Given two strings, find the shortest way of converting one to the other using
  - Insertion of a Character
  - Deletion of a Character
  - Substitution of a Character
- If all these processes cost 1, then this is the Levenshtein distance
- Important for approximate string matching, e.g. bio-informatics

Levenshtein Distance

- Example
  - university —> aniversity —> anniversity —> anniversaty —> anniversary

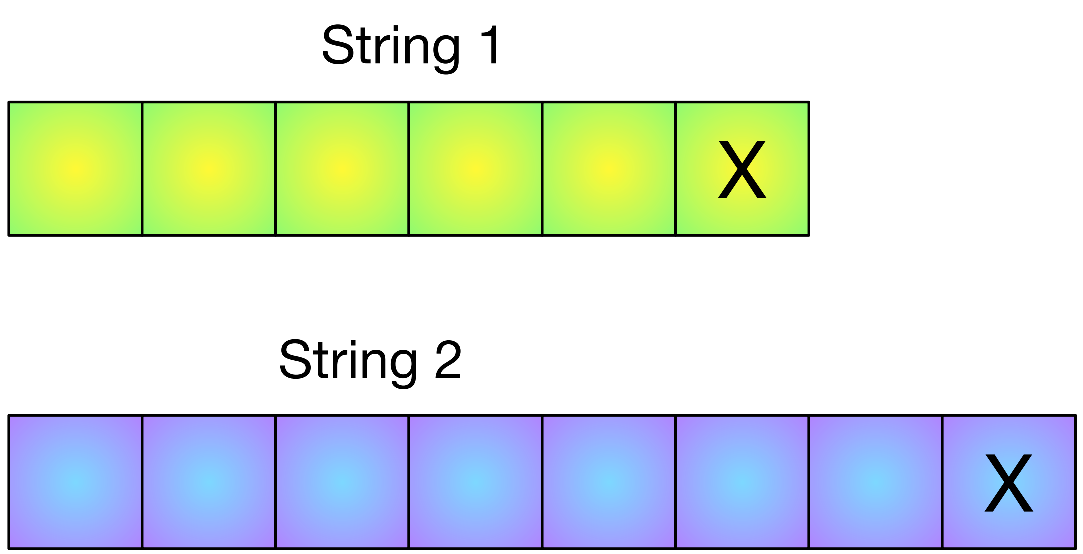
Levenshtein Distance

- Dynamic Programming approach
  - Define sub-problems:
    - Smaller strings:



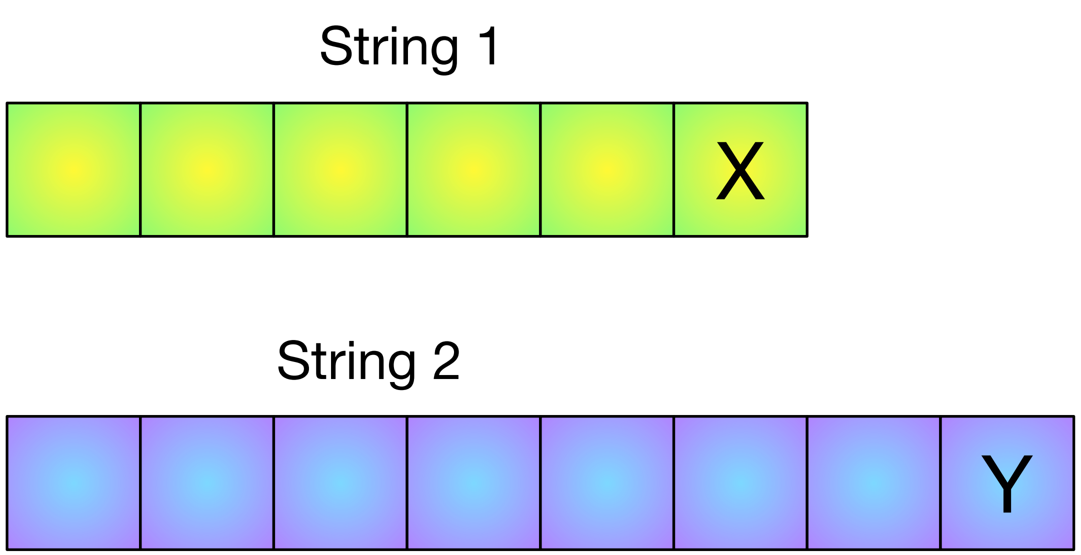
Levenshtein Distance

- Dynamic Programming Approach
  - Case 1a: Add the same letter
    - Distance does not change



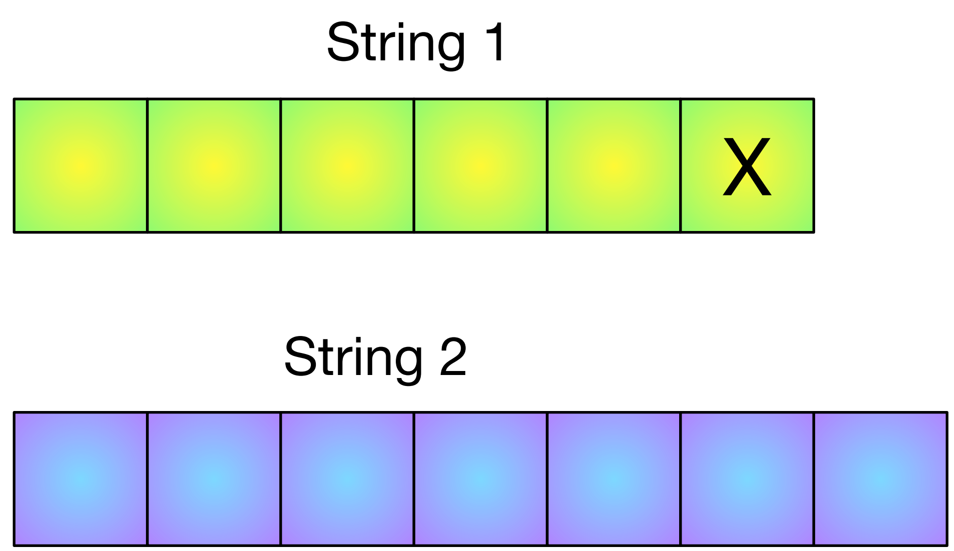
Levenshtein Distance

- Dynamic Programming
  - Case 1B: Add a different letter
    - Distance increases by one



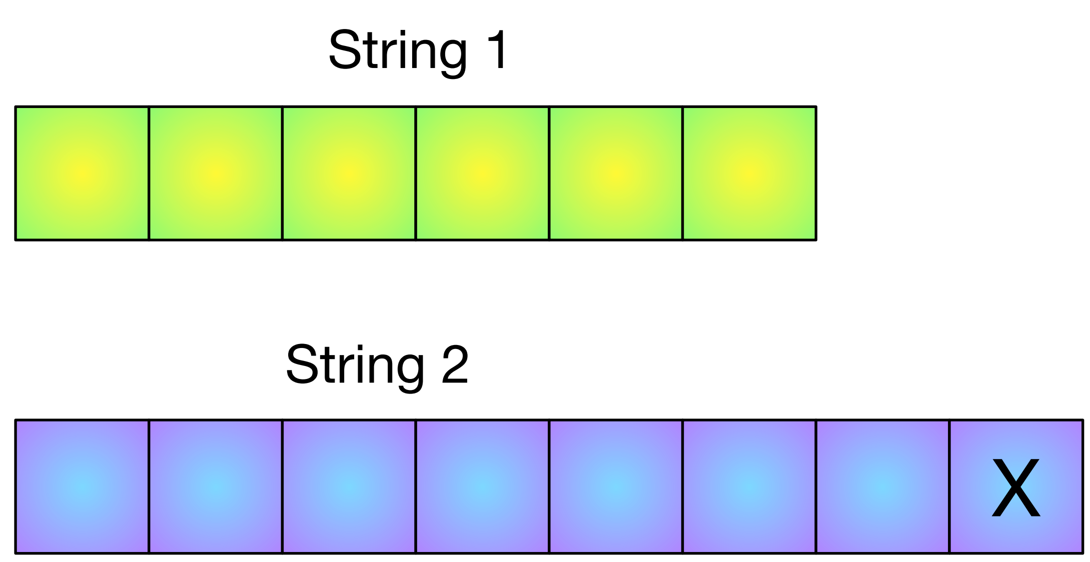
Levenshtein Distance

- Dynamic Programming Approach
  - Case 2: Add a letter to string 1
    - Increments distance



Levenshtein Distance

- Dynamic Programming Approach
  - Case 3:
    - Add a letter to String 2



Levenshtein Distance

- Dynamic Programming Approach
  - Strings are character arrays `A, B`
    - Look at sub-strings `delta(A[0::i], B[0::j])`
    - Use an indicator function

$$I(A[i] \neq B[j]) = \begin{cases} 1 & \text{if } A[i] \neq B[j] \\ 0 & \text{if } A[i] = B[j] \end{cases}$$

Levenshtein Distance

$$\delta(A[0 :: i], B[0 :: j])$$

$$= \min \left\{ \begin{array}{l} \delta(A[0 :: i-1], B[0 :: j]) + 1 \\ \delta(A[0 :: i], B[0 :: j-1]) \\ \delta(A[0 :: i-1], B[0 :: j-1] + I(A[i] \neq B[j]) \end{array} \right\}$$

Levenshtein Distance

- Dynamic Programming Approach
  - Base case
  - If one string is empty, then the distance is the length of the other string

Levenshtein Distance

- Dynamic Programming Approach
  - Bottom-up / Tabulation approach
    - Create a two dimensional table
      - Fill in first row and first column with length of other string
    - In order to find the edits without backtracking, mark where the value is coming from
      - ← ↑ ↖

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW
  - Fill in the first row / column

		O	S	L	O
	0	1	2	3	4
S	1				
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW
  - Fill in the first row / column

How to get from 'O' to 'S?'

$$\delta(A[0::2], B[0::2]) = ?$$

		O	S	L	O
	0	1	2	3	4
S	1	?			
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW
  - How to go from 'O' to 'S'?

$$\delta(A[0::2], B[0::2]) = ?$$

First choice: Go from "" to "S"; Add 'S' to the empty string:
$\delta(A[0::1], B[0::2]) + 1 = 2$

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	↑ ?			
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW
  - How to go from 'O' to 'S'

$\delta(A[0::2], B[0::2]) = ?$

Go from "" to "S"

Add 'S'

$\delta(A[0::2], B[0::1]) + 1 = 2$

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	← ?			
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW
  - How to go from 'O' to 'S'?

$\delta(A[0::2], B[0::2]) = ?$

Went from "" to ""

Now "O" to "S")
$\delta(A[0::1], B[0::1]) + 1 = 1$

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	?			
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW
  - How to go from 'O' to 'S'?
  - Starting with ' ' to ' '

$$\delta(A[0::2], B[0::2]) = ?$$

(Switch "O" to "S")

$$\delta(A[0::1], B[0::1]) + 1 = 1$$

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	↖ **1**			
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$$\delta(A[0 :: 3], B[0 :: 2]) = ?$$

- Reduce from:
  - 'OS' to 'S'
  - 'O' to ' '
  - 'O' to 'S'

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	**1**	**?**		
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$$\delta(A[0::3], B[0::2]) = ?$$

- Minimum of
  - 2+1
    - from above

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	**1**	**?**		
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$$\delta(A[0::3], B[0::2]) = ?$$

- Minimum of
  - 1+1
    - from left

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	1	?		
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$\delta(A[0::3], B[0::2]) = ?$

- Minimum of
  - 1+0
    - from upper left
    - 0 because the letters are the same

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	1	?		
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$$\delta(A[0::3], B[0::2]) = 1$$

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	1	1		
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$$\delta(A[0::4], B[0::2]) = ?$$

- Minimum of

Change / copy a letter

Add one letter

Drop a letter

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	1	1	?	
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$$\delta(A[0::4], B[0::2]) = ?$$

- Minimum of
  - 3+1
  - 2+1
  - 1+1

	A:	O	S	L	O
B:	0	1	2	3	4
S	1	1	1	← **2**	
N	2				
O	3				
W	4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$$\delta(A[0::4], B[0::2]) = ?$$

- Minimum of
  - 3+1
  - 2+1
  - 1+1

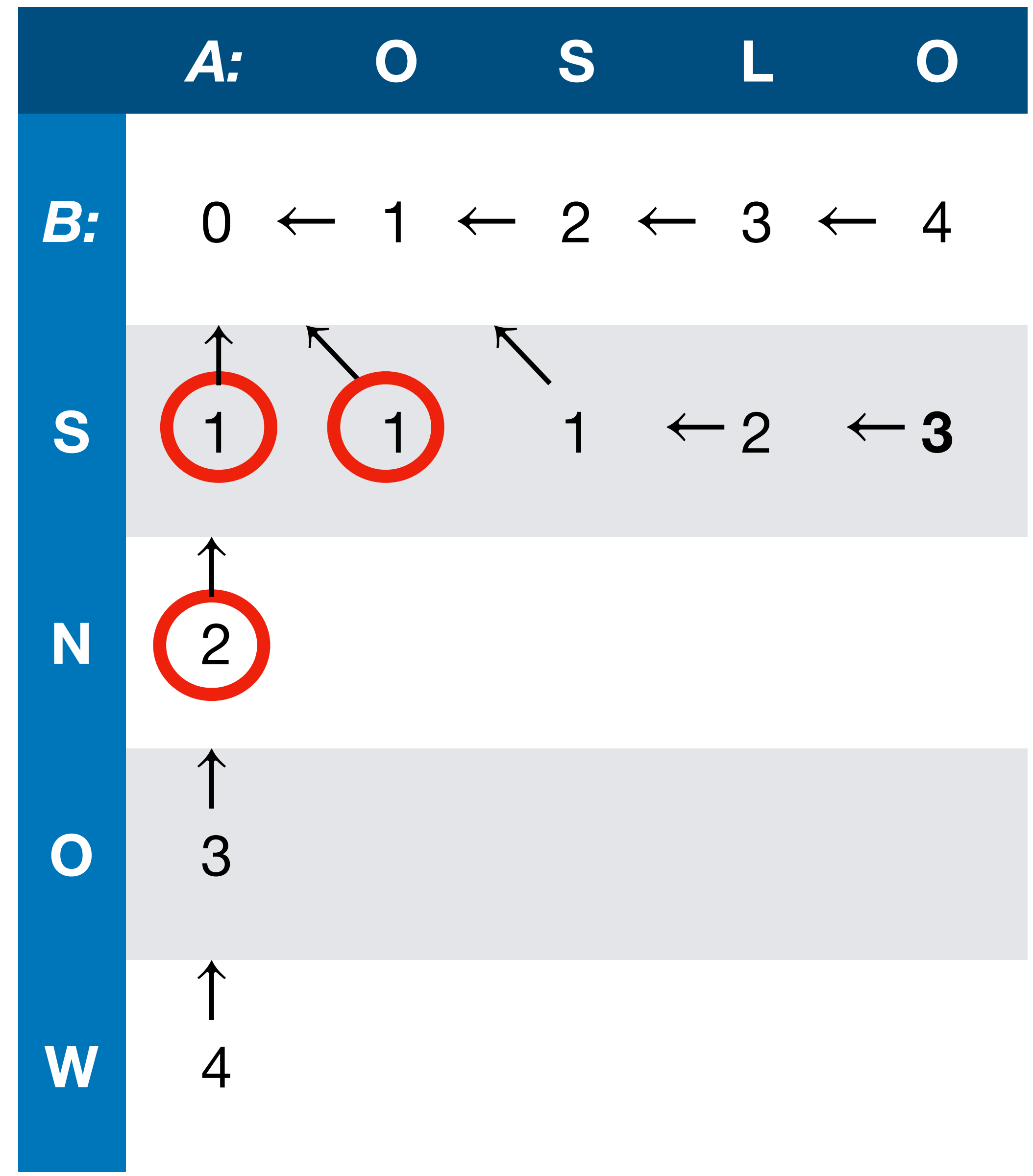
	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← **3**
N	↑ 2				
O	↑ 3				
W	↑ 4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$\delta(A[0::2], B[0::3]) = ?$

- Minimum of
  - 1+1
  - 2+1
  - 1+1
- which is 2



Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

$\delta(A[0::2], B[0::3]) = ?$

- Minimum of
  - 1+1
  - 2+1
  - 1+1
- which is 2

break the tie arbitrarily

	A:	**O**	**S**	**L**	**O**
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← **3**
N	↑ 2	↑ 2			
O	↑ 3				
W	↑ 4				

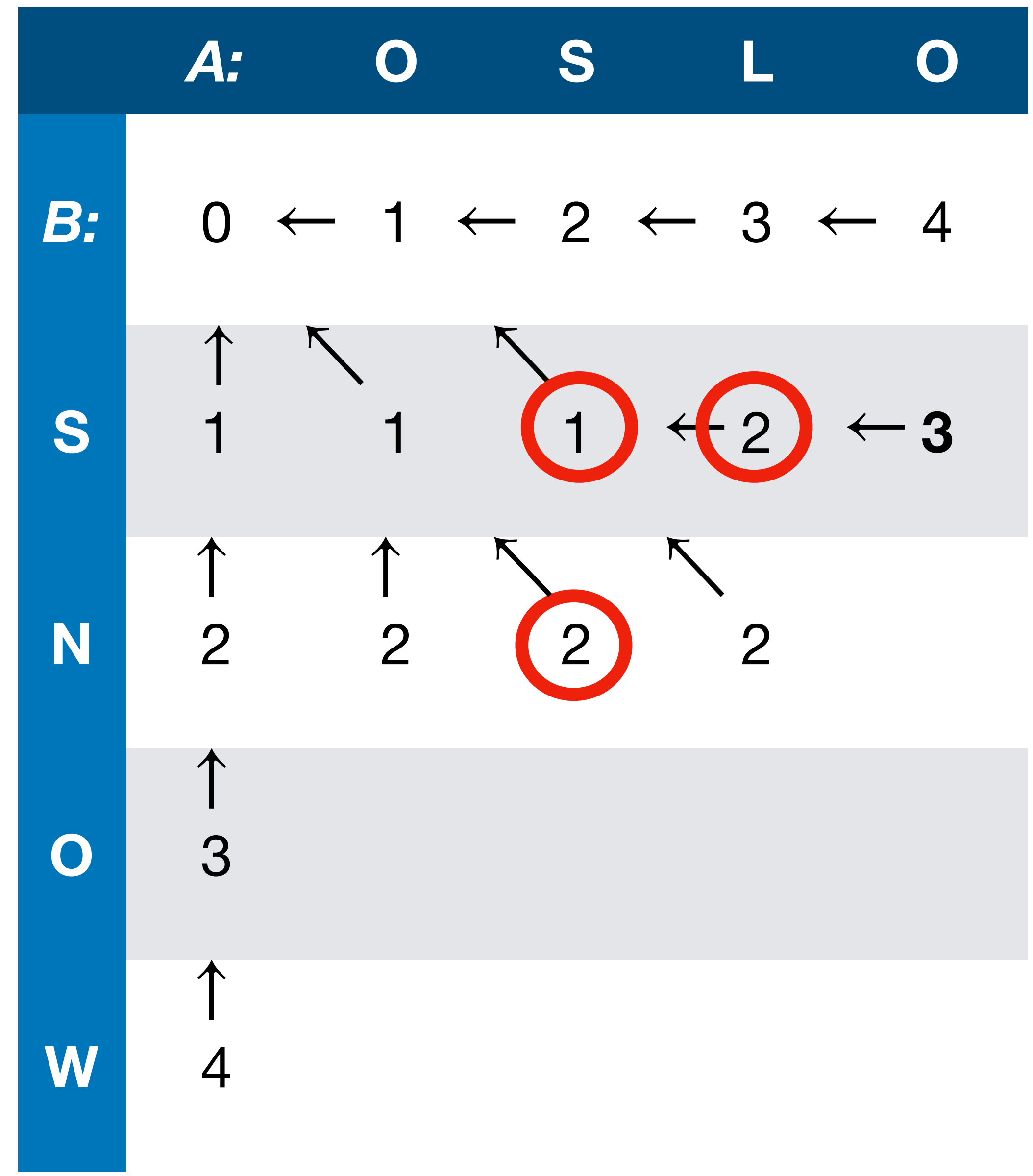
Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← **3**
N	↑ 2	↑ 2	↖ 2		
O	↑ 3				
W	↑ 4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW



Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3				
W	↑ 4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW
  - Because the letters are equal, changing a letter does not cost anything

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2			
W	↑ 4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW
  - Because the letters are equal, changing a letter does not cost anything

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3		
W	↑ 4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3	↖ 3	
W	↑ 4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3	↖ 3	↖ 2
W	↑ 4				

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3	↖ 3	↖ 2
W	↑ 4	↑ 3			

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3	↖ 3	↖ 2
W	↑ 4	↑ 3	↖ 3		

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3	↖ 3	↖ 2
W	↑ 4	↑ 3	↖ 3	↖ 4	

Levenshtein Distance

- Example
  - Edit distance between OSLO and SNOW

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3	↖ 3	↖ 2
W	↑ 4	↑ 3	↖ 3	↖ 4	↑ 3

Levenshtein Distance

- Interpreting the solution
  - Last step: add 'W'

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3	↖ 3	↖ 2
W	↑ 4	↑ 3	↖ 3	↖ 4	↑ 3

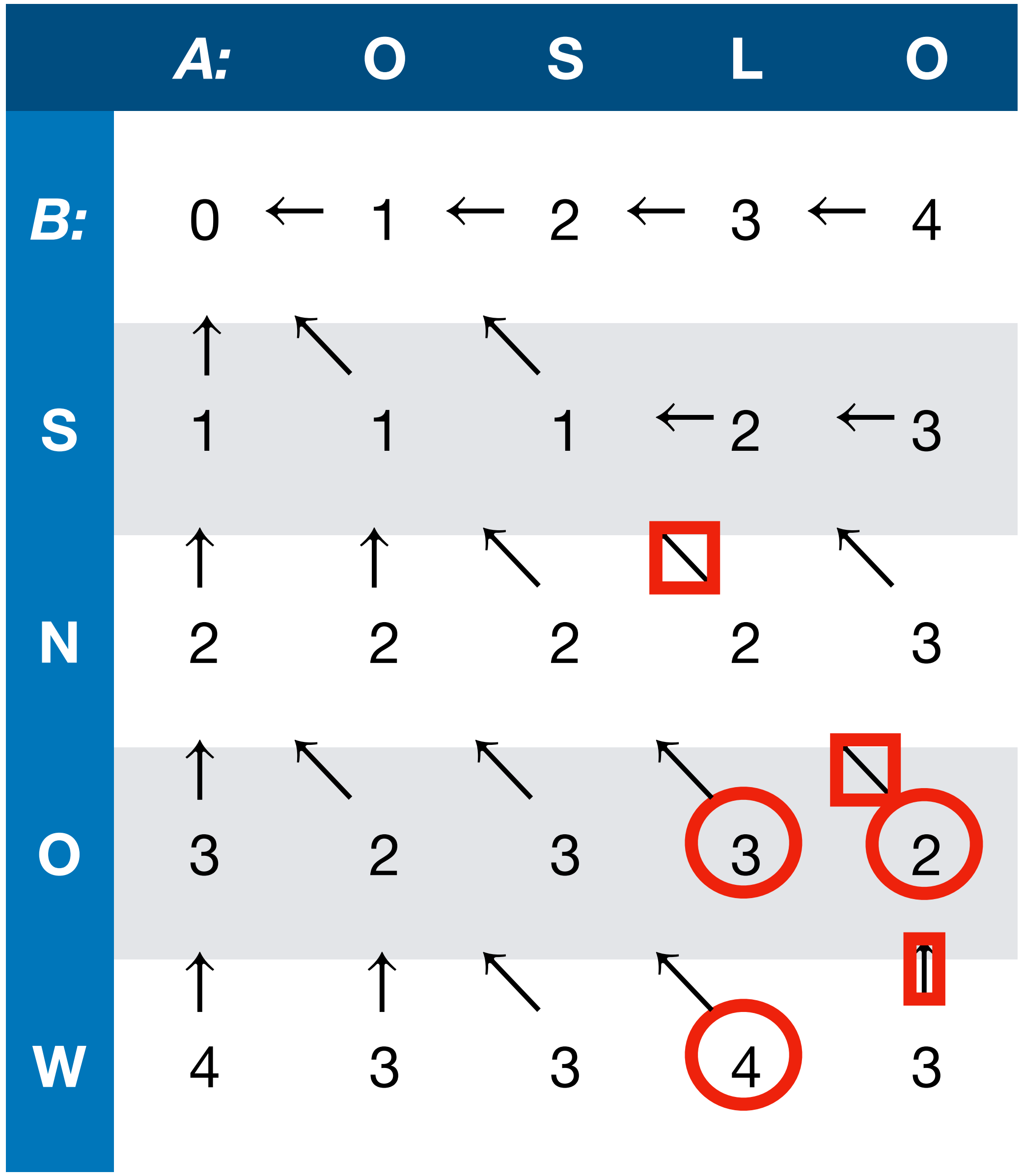
Levenshtein Distance

- Interpreting the solution
  - Last step:
    - add 'W'
  - Second last step:
    - Copy 'O'

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3	↖ 3	↖ 2
W	↑ 4	↑ 3	↖ 3	↖ 4	↑ 3

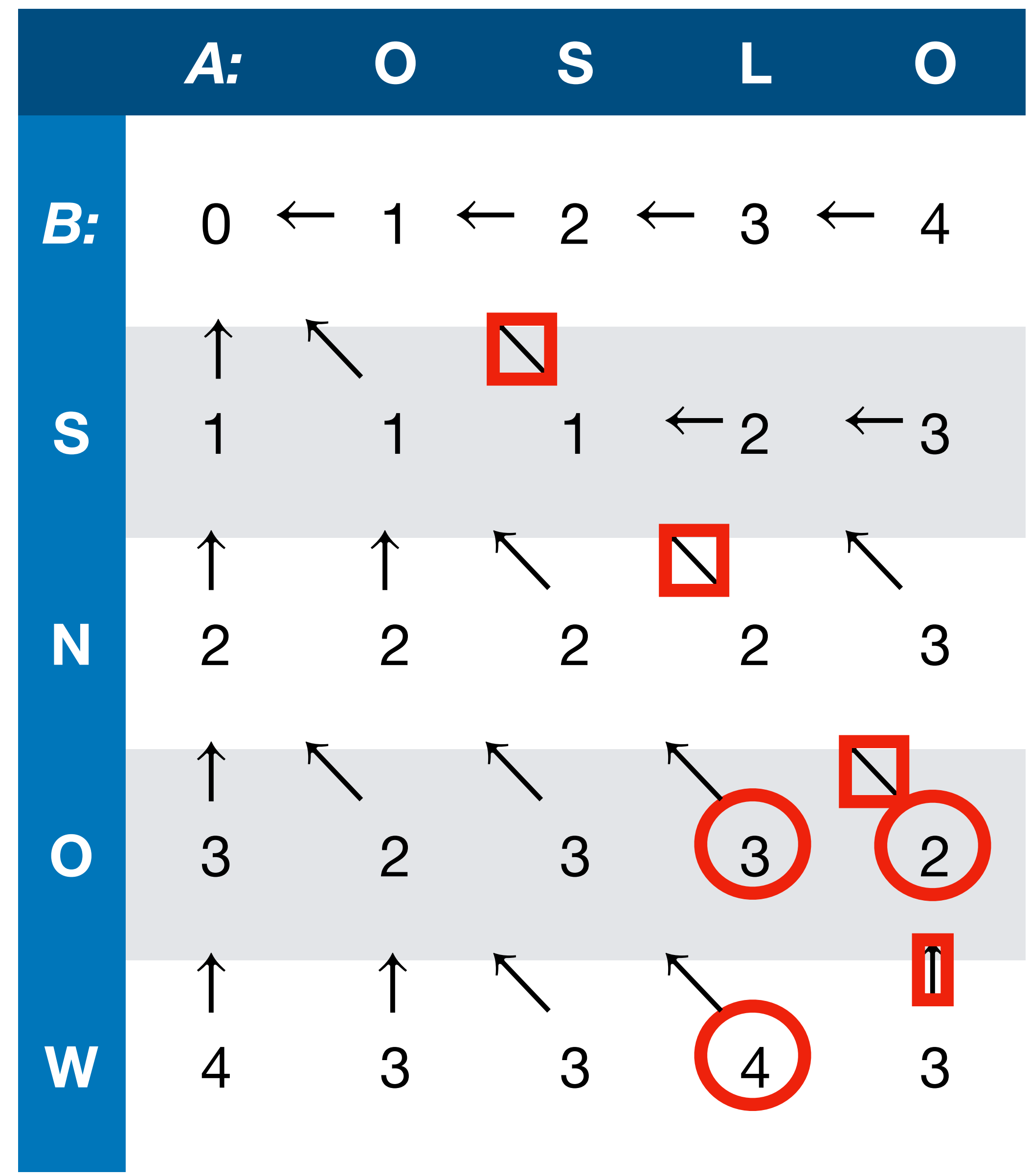
Levenshtein Distance

- Interpreting the solution
  - Last step:
    - add 'W'
  - Second last step:
    - Copy 'O'
  - Before
    - Change 'L' to 'N'



Levenshtein Distance

- Interpreting the solution
  - Last step:
    - add 'W'
  - Second last step:
    - Copy 'O'
  - Before
    - Change 'L' to 'N'
  - Before
    - Copy 'S'



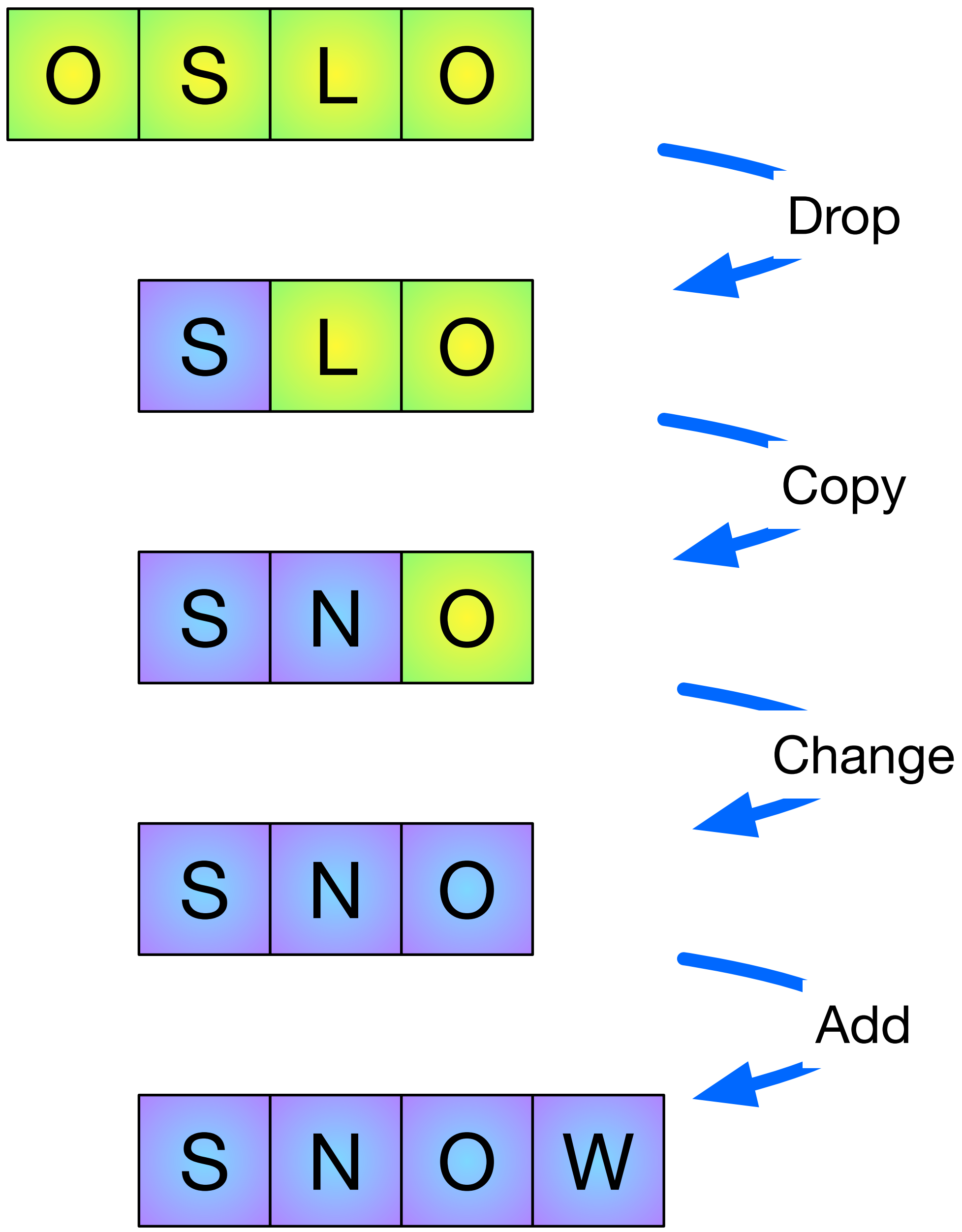
Levenshtein Distance

- Interpreting the solution
  - Last step:
    - add 'W'
  - Second last step:
    - Copy 'O'
  - Before
    - Change 'L' to 'N'
  - Before
    - Copy 'S'
  - Before
  - Drop 'O'

	A:	O	S	L	O
B:	0	← 1	← 2	← 3	← 4
S	↑ 1	↖ 1	↖ 1	← 2	← 3
N	↑ 2	↑ 2	↖ 2	↖ 2	↖ 3
O	↑ 3	↖ 2	↖ 3	↖ 3	↖ 2
W	↑ 4	↑ 3	↖ 3	↖ 4	↑ 3

Levenshtein Distance

- Interpreting the solution
  - Last step:
    - add 'W'
  - Second last step:
    - Copy 'O'
  - Before
    - Change 'L' to 'N'
  - Before
    - Copy 'S'
  - Before
  - Drop 'O'



Levenshtein Distance

- A Levenshtein Tableau has size $(n+1) \times (m+1)$
  - for string sizes n and m
- Filling in a tableau costs constant work
- Reconstructing the solution takes work $\leq n+m+2$

Levenshtein Distance

- We can simply change the formula to adjust to different edit models
  - For example: We can charge 2 for adding or dropping and 3 for changing a letter