Faculty of Information Technology,
Monash University

FIT2004: Algorithms and Data Structures

Week 6: Dynamic Programming

Overview

Divide and conquer (W 1-3)	Greedy algorithms (W 4-5)	Dynamic programming (W 6-7)	Network flow (W 8-9)	Data structures (W 10-11)

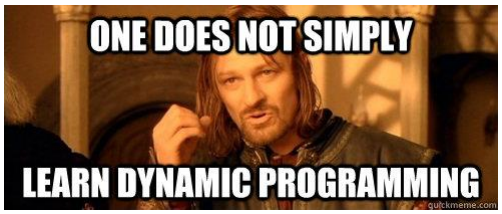
- Last Lecture: greedy algorithms
  - Dijkstra's Algorithm
  - Prim's Algorithm
  - Kruskal's Algorithm
- Today's Lecture
  - Introduction to Dynamic Programming
  - Coins Change
  - Unbounded Knapsack
  - 0/1 Knapsack
  - Edit Distance
  - Constructing Optimal Solution

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Dynamic Programming Paradigm

- A powerful optimization technique in computer science.
- Applicable to a wide-variety of problems that exhibit certain properties.
- Practice is the key to be good at dynamic programming.



Core Idea

- Divide a complicated problem by breaking it down into simpler subproblems in a recursive manner and solve these.
- Question: But how does this differ from `Divide and Conquer' approach?
  - **Overlapping subproblems**: the same subproblem needs to (potentially) be used multiple times (in contrast to independent subproblems in Divide and Conquer).
- We also need an **optimal substructure**: optimal solutions to subproblems help us find optimal solutions to larger problems.
- So how do we do this?
  - Identify the overlapping subproblems
  - Solve the smaller subproblems and **memoize** their solutions
  - use those memoized solutions to gradually build solution for the original problem

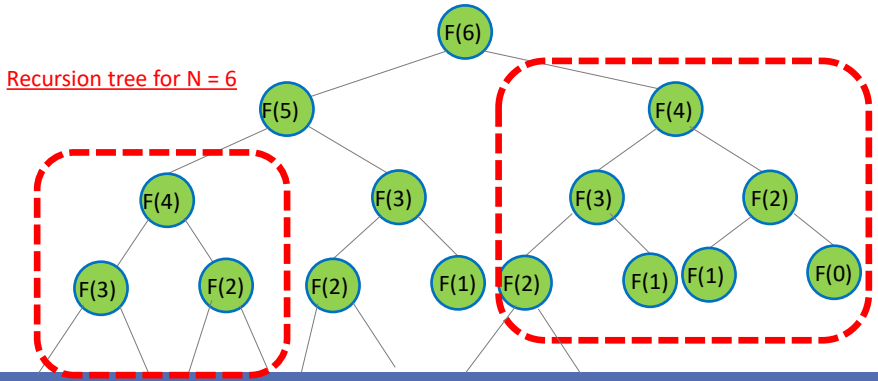
Memoize means **storing the result so that they can be used next time instead of calculating the same thing again and again.

N-th Fibonacci Number

```
F(N)
    if N == 0 or N == 1
        return N
    else
        return F(N - 1) + F(N - 2)
```

Time Complexity
T(1) = b // b and c are constants
T(N) = T(N-1) + T(N-2) + c
= $O(2^N)$

Recursion tree for N = 6



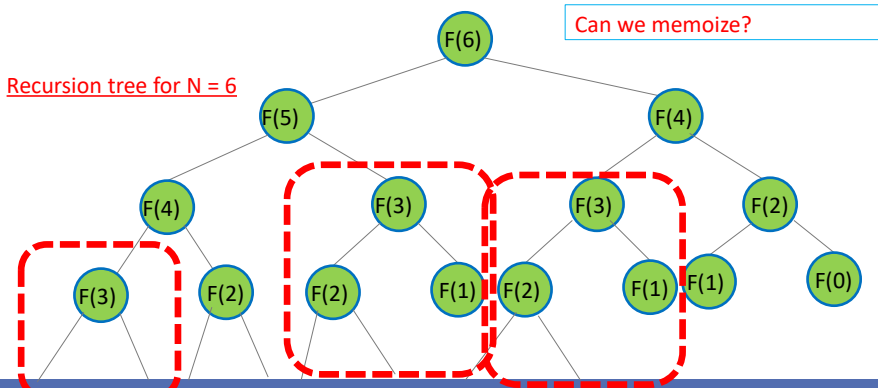
N-th Fibonacci Number

```
F(N)
    if N == 0 or N == 1
        return N
    else
        return F(N - 1) + F(N - 2)
```

Time Complexity
T(1) = b // b and c are constants
T(N) = T(N-1) + T(N-2) + c
= $O(2^N)$

Can we memoize?

Recursion tree for N = 6



Fibonacci with Memoization: Version 1

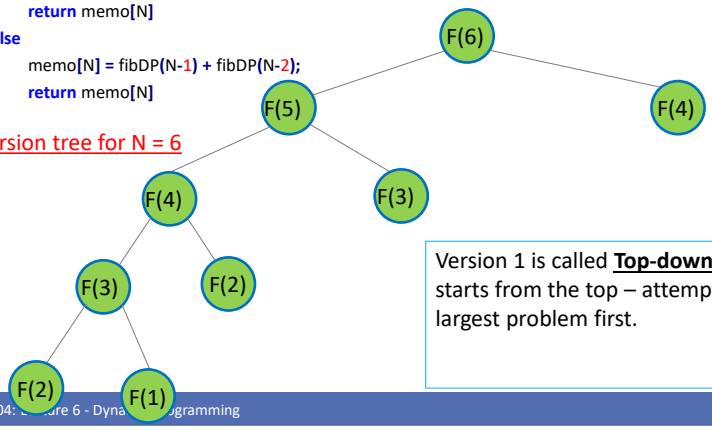
```
memo[0] = 0  // 0th Fibonacci number
memo[1] = 1  // 1st Fibonacci number
for i=2 to i=N:
    memo[i] = -1
fibDP(N)
    if memo[N] != -1
        return memo[N]
    else
        memo[N] = fibDP(N-1) + fibDP(N-2);
        return memo[N]
```

Time Complexity

calls fibDP() roughly 2*N times

So the complexity is O(N)

Recursion tree for N = 6



Version 1 is called **Top-down** because it starts from the top – attempting the largest problem first.

Fibonacci with Memoization: Version 2

```
memo[0] = 0   // 0th Fibonacci number
memo[1] = 1   // 1st Fibonacci number
for i=2 to i=N:
        memo[i] = memo[i-1] + memo[i-2]
```

Time Complexity

O(N)

memo	0	1	1	2	3	5	8	13	21	34

Version 2 is called **Bottom-up** because it starts from the bottom – solving the smallest problem first.

Dynamic Programming Strategy

1. Assume you already know the solutions of all sub-problems and have memoized these solutions (overlapping subproblems).
   - E.g., Assume you know Fib(i) for every i < n.
2. Observe how you can solve the original problem using memoized solutions (optimal substructure) .
   - E.g., Fib(n) = Fib(n-1) + Fib(n-2).
3. Solve the original problem by building upon solutions to the sub-problems.
   - E.g., Fib(0), Fib(1), Fib(2), …, Fib(n).

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Coins Change Problem

Problem: A country uses N coins with denominations $\{a_1, a_2, \ldots, a_N\}$. Given a value V, find the minimum number of coins that add up to V.

- **Example:** Suppose the coins are {1, 5, 10, 50} and the value V is 110. The minimum number of coins required to make 110 is 3 (two 50 coins, and one 10 coin).
- Greedy solution does not always work. E.g., If Coins = {1, 5, 6, 9}. What would be the minimum number of coins to make 12?
  - It would be 4 (i.e. coins of 9, 1, 1, 1)- a greedy solution.
  - However, the optimal is 2 (i.e., two 6 coins).
- Similarly, what is the minimum number of coins to make 13?
  - 3 (i.e., two 6 coins and one 1 coin)- optimal vs 5- greedy.

Coins Change Problem

- **Problem:** A country uses N coins with denominations $\{a_1, a_2, \ldots, a_N\}$. Given a value V, find the minimum number of coins that add up to V.
- Overlapping subproblems: What shall we store in the memo array?
- We want to know the minimum number of coins which add up to V.
- So lets try

MinCoins[v] = {The fewest coins required to make exactly \$$v$}

- Note: your first guess to include in the memo array may not be right, so try the most obvious thing and then play around if you can't make it work.

Coins Change Problem

- **Problem:** A country uses N coins with denominations $\{a_1, a_2, \ldots, a_N\}$. Given a value V, find the minimum number of coins that add up to V.
- Overlapping subproblems:

  MinCoins[v] = {The fewest coins required to make exactly \$$v$}
- Optimal substructure: To find the optimal substructure, we first deal with the base case(s). In this case, to make \$0 requires 0 coins, so MinCoins[0] = 0.
- Assume we have optimal solutions for all $v < V$ (stored in MinCoins[0..V-1]). How could we determine MinCoins[V]?

Coins Change Problem - Example

Coins: [9, 6, 5, 1]

V: 12

MinCoins:

0	1	2	3	4	1	1	2	3	1	2	2	
0	1	2	3	4	5	6	7	8	9	10	11	12

- What options do we have to try and make V = \$12?
- We have to use any available coin! Lets try using the 9…
- After choosing coin 9, what would be the optimal thing to do?
- Look at MinCoins[12-9] since now we need to make the other \$3, and we already know the best way to do that.
- Repeat this idea for the other coins and see which is best.

Coins Change Problem - Example

Coins: [9, 6, 5, 1]

V: 12

MinCoins:

0	1	2	3	4	1	1	2	3	1	2	2	
0	1	2	3	4	5	6	7	8	9	10	11	12

- MinCoins[12] =

  1 + **min**(MinCoins[12-9], MinCoins[12-6], MinCoins[12-5], MinCoins[12-1])

  = 1 + **min**(MinCoins[3], MinCoins[6], MinCoins[7], MinCoins[11])

  = 1 + **min**(3, 1, 2, 2)

  = 2

- In general, MinCoins[*v*] = 1 + min(MinCoins[*v*-*c*] for all c in coins, where *c* <= *v*)
- Note also that if the value is less than every coin, then it cannot be made. This can be ignored if there is a $1 coin.

Coins Change Problem - Example

Coins: [9, 5, 6, 1]

V: 12

MinCoins:

0	1	2	3	4	1	1	2	3	1	2	2	2
0	1	2	3	4	5	6	7	8	9	10	11	12

Overlapping subproblems:

MinCoins[v] = {The fewest coins required to make exactly $*v*}.

Optimal substructure:

$$\text{MinCoins[v]} = \begin{cases} 0 & \text{if } v = 0, \\ \infty & \text{if } v > 0 \text{ and } v < c[i] \text{ for all } i, \\ \min\limits_{\substack{1 \le i \le n \\ c[i] \le v}} (1 + \text{MinCoins[v - c[i]]}) & \text{otherwise} \end{cases}$$

Coins Change Problem – Implementation (Bottom up)

With DP, you can generally implement straight from the recurrence to code.

$$\text{MinCoins}[v] = \begin{cases} 0 & \text{if } v = 0, \quad \text{A} \\ \infty & \text{if } v > 0 \text{ and } v < c[i] \text{ for all } i \quad \text{B} \\ \min\limits_{\substack{1 \le i \le n \\ c[i] \le v}} (1 + \text{MinCoins}[v - c[i]]) & \text{otherwise} \quad \text{C} \end{cases}$$

Coin_change(c[1..n], V)
  Mincoins[0..V] = ∞ (note we start from 0 index here)
  Mincoins[0] = 0 *(from recurrence)* A
  for each i from 1 to V
    options = [ ]
    for k from 1 to n
      if i < c[k] *(from recurrence)* B
        do nothing
      else
        options.append(…) = [MinCoins[i-c[k]], provided i >= c[k]] *(from recurren* C
    Mincoins[i] = min(options)+1
  return Mincoins[V]

Coins Change – Implementation (top down)

//Assume that the *coin_change* function has appropriately initialised memo to an
//array of nulls, and called our auxiliary function

Coin_change_aux(*c*[1..*n*], *v*)
  if *v* = 0, return 0
  if memo[*v*] = null
    min_coins = infinity
    for *i* in 1 to *n*
      if *c*[*i*] <= *v*
        min_coins = min(min_coins, 1 + *coin_change_aux(c, v-c[i])*)
    memo[*v*] = min_coins
  return memo[*v*]

This recursive call just returns memo[v-c[i]] instantly if we have already calculated it (because of this "if")

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Unbounded Knapsack Problem

Problem: Given a capacity *C* and a set of items with their weights and values, you need to pick items such that their total weight is at most *C* and their total value is maximized. What is the maximum value you can take? In unbounded knapsack, you can pick an item as many times as you want.

Item	A	B	C	D
Weight	9kg	5kg	6kg	1kg
Value	$550	$350	$180	$40

Example: What is the maximum value for the above example given the capacity is 12 kg?

Answer: $780 (take two Bs and two Ds).

Greedy solution does not always work.

This is one of the most popular algorithmic problems globally!

DP Solution for Unbounded Knapsack

Problem: Given a capacity *C* and a set of items with their weights and values, you need to pick items such that their total weight is at most *C* and their total value is maximized. What is the maximum value you can take? In unbounded knapsack, you can pick an item as many times as you want.

- We want the LARGEST value under a given weight.
- Overlapping subproblems: What shall we store in the memo array?
  - Memo[*i*] = LARGEST value with capacity at most *i*.
- Optimal substructure: If we know optimal solutions to all subproblems, how can we build an optimal solution to a larger problem?
  - Similar logic to coin change:
    - We need to choose an item…
    - For each possible item choice, find out how much value we could get (using subproblems) and then take the best one.

DP Solution for Unbounded Knapsack

- What is the maximum value given capacity is 12 kg?
- If we take item 1, then we have 3kg left.
- The best we can do with 3kg is memo[3] = $120.
- So one option for memo[12] would be value[1] + memo[12-weight[item_1]].

Item	1	2	3	4
Weight	9kg	5kg	6kg	1kg
Value	$550	$350	$180	$40

Memo

40	80	120	160	350	390	430	470	550	700	740	
1	2	3	4	5	6	7	8	9	10	11	12

DP Solution for Unbounded Knapsack

Memo[12] could be:

- value[1] + memo[12-weight[item_1]] = 550 + 120 = 670
- value[2] + memo[12-weight[item_2]] = 350 + 430 = 780
- value[3] + memo[12-weight[item_3]] = 390 + 180 = 570
- value[4] + memo[12-weight[item_4]] = 740 + 40 = 780

- Choose the best!

Item	1	2	3	4
Weight	9kg	5kg	6kg	1kg
Value	$550	$350	$180	$40

Memo	40	80	120	160	350	390	430	470	550	700	740	
	1	2	3	4	5	6	7	8	9	10	11	12

DP Solution for Unbounded Knapsack

Memo[12] could be:

- value[1] + memo[12-weight[1]] = 550 + 120 = 670
- value[2] + memo[12-weight[2]] = 350 + 430 = 780
- value[3] + memo[12-weight[3]] = 390 + 180 = 570
- value[4] + memo[12-weight[4]] = 740 + 40 = 780

- Choose the best!

Item	1	2	3	4
Weight	9kg	5kg	6kg	1kg
Value	$550	$350	$180	$40

Memo	40	80	120	160	350	390	430	470	550	700	740	780
	1	2	3	4	5	6	7	8	9	10	11	12

DP Solution for Unbounded Knapsack

Lets write our recurrence:

- What is our **base case**?
  - With no capacity, we cannot take any items. Also note, as before, that if an item is heavier than the capacity we have left, we cannot take it.
- Otherwise, we want the maximum over all values ($1 \le i \le n$, v_i) of items that we could take ($w_i \le c$).
- But also taking into account the **optimal value** we could fit into the rest of our knapsack, once we took that item.
- Complete the **recurrence**:

$$\text{MaxValue}[c] = \begin{cases} 0 & \text{if } c < w_i \text{ for all } i \\ \max_i(v_i + ??????) & \text{otherwise} \end{cases}$$

DP Solution for Unbounded Knapsack

Overlapping subproblems: memo[*i*] = most value with capacity at most *i*.

Optimal substructure:

$$\text{MaxValue}[c] = \begin{cases} 0 & \text{if } c < w_i \text{ for all } i \\ \max_i(v_i + ??????) & \text{otherwise} \end{cases}$$

$$\text{MaxValue}[c] = \begin{cases} 0 & \text{if } c < w_i \text{ for all } i, \\ \max\limits_{\substack{1 \le i \le n \\ w_i \le c}} (v_i + \text{MaxValue}[c - w_i]) & \text{otherwise.} \end{cases}$$

Bottom-up Solution

```
// Construct Memo[ ] starting from 1 until C in a way similar to previous slide.
Initialize Memo[ ] to contain 0 for all indices
for c = 1 to C
   maxValue = 0
   for i=1 to N
      if Weight[ i ] <= c
         thisValue = Value[i]  + Memo[c - Weight[ i ] ]
         if thisValue > maxValue
            maxValue = thisValue
   Memo[c] = maxValue
```

Time Complexity:
O(NC)
Space Complexity:
O(C + N)

E.g., Fill Memo[13]

Item	1	2	3	4
Weight	9kg	5kg	6kg	1kg
Value	$550	$350	$180	$40

Memo	40	80	120	160	350	390	430	470	550	700	740	780	
	1	2	3	4	5	6	7	8	9	10	11	12	13

Top-down Solution

```
Initialize Memo[ ] to contain -1 for all indices // -1 indicates solution for this index has not yet been computed
Memo[0] = 0
function knapsack(Capacity)
   if Memo[Capacity] != -1:
      return Memo[Capacity]
   else:
      maxValue = 0
      for i=1 to N
         if Weight[ i ] <= Capacity
            thisValue = Value[i]  + knapsack(Capacity - Weight[ i ] )
            if thisValue > maxValue
               maxValue = thisValue
      Memo[Capacity] = maxValue
      return Memo[Capacity]
```

Bottom up solution:

Values[i] + Memo[Capacity – Weights[i]]

Top Down vs Bottom Up

- Top-down **may** save some computations (E.g., some smaller subproblems may not need to be solved).
- Space saving trick may be applied for bottom-up to reduce space complexity.
- You may find one easier to think about.
- In some cases, the solution cannot be written bottom-up without some silly contortions.

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

0/1 Knapsack Problem

Same as unbounded knapsack except that each item can only be picked at most once.

Example: What is the maximum value for the example given below given capacity is 11kg?

Answer: $590 (B and D) for capacity of 10kg.
Greedy solution may not always work.

(Note that A & C together can produce 11kg, but value is only $580)

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

0/1 Knapsack Problem

Same as unbounded knapsack except that each item can only be picked at most once.

Difference from unbounded: If we pick an item X, giving us a remaining capacity R, we have to somehow make sure that X is not part of the optimal solution to our new subproblem of size R.

Idea: Lets have two axes on which we think about subproblems:

- Capacity
- Which items are part of the subproblem

0/1 Knapsack Problem

Problem: What is the solution for 0/1 knapsack for items {A,B,C,D} where capacity = 11kg.

Assume that we have computed solutions for every capacity<=11 considering the items {A,B,C} (see table below).

What is the solution for capacity=11 and set {A,B,C,D} ?

- **Case 1:** the knapsack must **NOT** contain D
  - Solution for 0/1 knapsack with set {A,B,C} and capacity 11 = 580
- **Case 2:** the knapsack **must** contain D
  - The value of item D + solution for 0/1 knapsack with set {A,B,C} and capacity 11-9=2
  - This gives a value of 550+40 = 590
- **Solution = max(Case1, Case2)**

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

	1	2	3	4	5	6	7	8	9	10	11
{A,B,C}	40	40	40	40	350	390	390	390	390	390	580

	1	2	3	4	5	6	7	8	9	10	11
{A,B,C,D}	40	40	40	40	350	390	390	390	550	590	**590**

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]

Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

max(40,0)

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	?											
3	C												
4	D												

i

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590		

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

max(

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590		

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

max(580

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590		

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

max(580, 550 + 40)

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590		

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

max(580, 550 + 40)

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590	590	

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

max(

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590	590	

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

max(620

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590	590	

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

max(620, 550 + 40)

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590	590	

0/1 Knapsack Problem

Assume we know the optimal solutions for every subproblem and results are stored in Memo[ ][ ]
Memo[i][c] contains the solution of knapsack for **Set[1 … i]** and capacity **c**.

Item	A	B	C	D
Weight	6kg	1kg	5kg	9kg
Value	$230	$40	$350	$550

max(620, 550 + 40)

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590	590	620

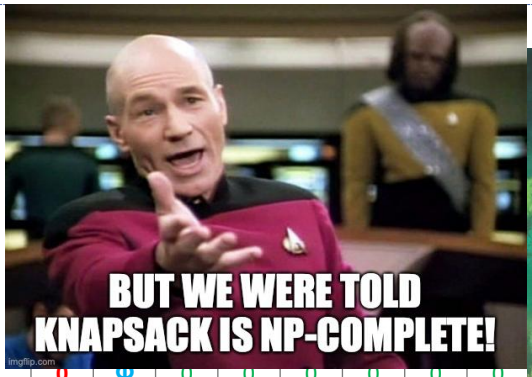
0/1 Knapsack Problem

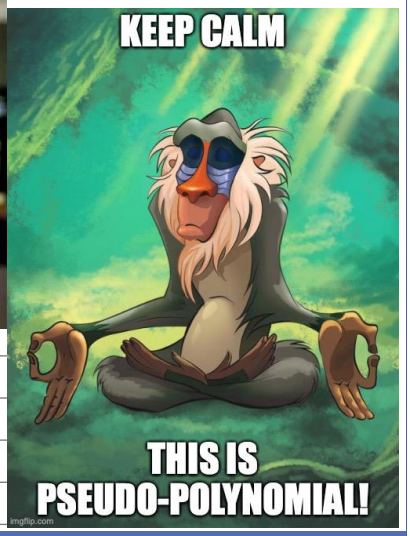
Complexity:

- We need to fill this two-dimensional grid.
- Filling each cell is O(1) since it is the max of 2 numbers, each of which can be computed in a constant number of lookups.
- Therefore, the time and space complexity are both O(*NC*) where *N* is the number of items and *C* is the capacity of the knapsack.

		1	2	3	4	5	6	7	8	9	10	11	12
0	Φ	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	230	230	230	230	230	230	230
2	B	40	40	40	40	40	230	270	270	270	270	270	270
3	C	40	40	40	40	350	390	390	390	390	390	580	620
i 4	D	40	40	40	40	350	390	390	390	550	590	590	620

0/1 Knapsack Problem





i								
	0	Φ	0	0	0	0	0	0
	1	A	0	0	0	0	0	230
	2	B	40	40	40	40	40	230
	3	C	40	40	40	40	350	390
	4	D	40	40	40	40	350	390

0/1 Knapsack Problem

Complexity:

- Knapsack complexity is not polynomial, it is actually exponential
- The knapsack algorithm runs in O(*CN*), where
  - N is the # of items (array)
  - C is the max capacity (single value)
- What is the definition of time complexity?
  - The amount of time taken by an algorithm to run, as a function of the length of the input (in bits) (not the value of the input).
- Think about how many bits it takes to specify input.
- Instead of *C*, lets talk about *B*, the number of bits to specify *C*.
- *C* is the capacity. *C* can be described with $\log_2 C + 1$ bits.
  - For C = 4kg, # of bits = 3 (C = 100 in binary)
  - For C = 8kg, # of bits = 4 (C = 1000 in binary)
  - For C = 16kg, # of bits = 5 (C = 10000 in binary)
- $\log_2 C + 1 = B \Rightarrow C = 2^B$
- Now we can say our algorithm runs in O(*CN*) = O($2^B N$), which is not polynomial in the size of the input (as expected for an NP-complete problem).

Reducing Space Complexity

- While generating each row, we only need to look at values from the previous row.
- So all values from the earlier rows may be discarded.
- Reduces space complexity to O(C) (or O(2^B) as we saw).

- **Note:** Space saving not possible for top-down dynamic programming (since we don't know the order we solve subproblems).
- **Note:** This simple space saving trick **cannot be used when we want to reconstruct the solution**. We need a more advanced idea for that case!

		1	2	3	4	5	6	7	8	9	10	11	12
3	C	40	40	40	40	350	390	390	390	390	390	580	620
4	D	40	40	40	40	350	390	390	390	550	590		

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Edit Distance

- The words computer and commuter are very similar, and a change of just one letter, p → m, will change the first word into the second.
- The word sport can be changed into sort by the deletion of p, or equivalently, sort can be changed into sport by the insertion of p.
- Notion of editing provides a simple and handy formalisation to compare two strings.
- The goal is to convert the first string (i.e., sequence) into the second through a series of edit operations.
- The permitted edit operations are:
  1. insertion of a symbol into a sequence.
  2. deletion of a symbol from a sequence.
  3. substitution or replacement of one symbol with another in a sequence.

Edit Distance

Edit distance between two sequences

- Edit distance is the minimum number of edit operations required to convert one sequence into another (*assuming that all operations have equal costs*).

For example:

- Edit distance between computer and commuter is 1.
- Edit distance between sport and sort is 1.
- Edit distance between shine and sings is ?
- Edit distance between dnasgivethis and dentsgnawstrims is ?

Examples

shine → sings
s = s (no change)
h → i (substitute 'h' → 'i')
i = n (substitute 'i' → 'n')
n → g (substitute 'n' → 'g')
e → s (substitute 'e' → 's')
4 edits (substitutes) in total

shine → sings
s = s (no change)
Delete 'h'
i = i (no change)
n = n (no change)
e → g (substitute 'e' → 'g')
Insert 's'
3 edits (1 delete, 1 substitute, 1 insert) in total

dnasgivethis → dentsgnawstrims
d = d (no change)
Insert 'e'
n = n (no change)
a → t (substitute 'a' → 't')
s = s (no change)
g = g (no change)
Insert 'n'
i = a (substitute 'i' → 'a')
v → w (substitute 'v' → 'w')
e → s (substitute 'e' → 's')
t = t (no change)
h → r (substitute 'h' → 'r')
i = i (no change)
Insert 'm'
s = s (no change)
3 inserts + 5 substitutes = 4 edits in total

Some Applications of Edit Distance

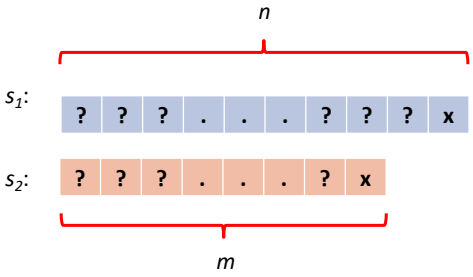
- Natural Language Processing
  - Auto-correction
  - Query suggestions
- Bioinformatics
  - DNA/Protein sequence alignment

Computing Edit Distance

We want to convert s_1 to s_2 containing n and m letters, respectively.

To gain an intuition for this problem, lets look at some situations we might run into.

This is a good technique in general, try playing around with the problem and see what happens.



How much does it cost to turn s_1 into s_2 if the last characters are the same?

We can leave the last character, and just convert the front part of one string into the front part of the other

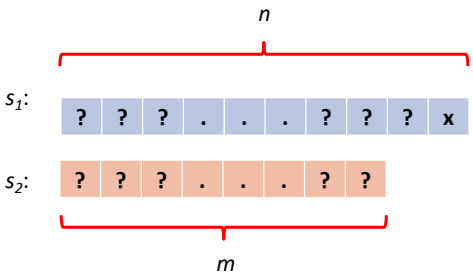
edit(s_1[1..n], s_2[1..m])
=edit(s_1[1..n-1], s_2[1..m-1])

Computing Edit Distance

We want to convert s_1 to s_2 containing n and m letters, respectively.

To gain an intuition for this problem, lets look at some situations we might run into.

This is a good technique in general, try playing around with the problem and see what happens.



How much does it cost to turn s_1 into s_2 if the last characters are different?

We have some options.

Computing Edit Distance

- Remember! We can assume that we have solved ALL subproblems already. In other words, we know:
  - Edit(s_1[1..i], s_2[1..j]) for all $i \leq n$, $j \leq m$ BUT NOT when $i = n$ AND $j = m$ (since this is the exact problem we are trying to solve).
- Alternatively, we could think about it visually.
- In this table, cell[i][j] is the cost of turning s_1[1..i] into s_2[1..j]

	1	2	3	.	.	.	m
1							
2							
3							
.							
.							
.							
n							

Known:
Unknown:

Computing Edit Distance

We know Edit(s_1[1..i], s_2[1..j]) for all $i \leq n$, $j \leq m$ BUT NOT $i = n$ AND $j = m$.

Equivalently, we know all the blue cells.

	1	2	3	.	.	.	m
1							
2							
3							
.							
.							
.							
n							

From the bottom right corner, there are three things we can do:

- Go up
- Go left
- Go left and up

Computing Edit Distance

We know Edit(s_1[1..i], s_2[1..j]) for all $i \leq n$, $j \leq m$ BUT NOT $i = n$ AND $j = m$.

Equivalently, we know all the blue cells.

	1	2	3	.	.	.	m
1							
2							
3							
.							
.							
.							
n							

From the bottom right corner, there are three things we can do:

- Go up
- Go left
- Go left and up

Going up:
Subproblem: edit(s_1[1..n-1], s_2[1..m]).

- First delete s_1[n].
- Then turn s_1[1..n-1] into s_2[1..m].

Total cost:
cost(delete) + edit(s_1[1..n-1], s_2[1..m]).

Computing Edit Distance

We know Edit(s_1[1..i], s_2[1..j]) for all $i \leq n$, $j \leq m$ BUT NOT $i = n$ AND $j = m$.

Equivalently, we know all the blue cells.

	1	2	3	.	.	.	m
1							
2							
3							
.							
.							
.							
n							

From the bottom right corner, there are three things we can do:

- Go up
- Go left
- Go left and up

Going left:
Subproblem: edit(s_1[1..n], s_2[1..m-1]).

- First turn s_1[1..n] into s_2[1..m-1].
- Then insert s_2[m] at the end of s_1.

Total cost:
edit(s_1[1..n], s_2[1..m-1]) + cost(insert).

Computing Edit Distance

We know Edit(s_1[1..i], s_2[1..j]) for all $i \leq n$, $j \leq m$ BUT NOT $i = n$ AND $j = m$.

Equivalently, we know all the blue cells.

	1	2	3	.	.	.	m
1							
2							
3							
.							
.							
.							
n							

From the bottom right corner, there are three things we can do:

- Go up
- Go left
- Go left and up

Going left:
Subproblem: edit(s_1[1..n-1], s_2[1..m-1]).

- Replace s_1[n] with s_2[m].
- Turn s_1[1..n-1] into s_2[1..m-1].

Total cost:
edit(s_1[1..n-1], s_2[1..m-1]) + cost(replace).

Computing Edit Distance

Base cases?

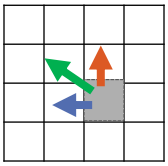
- When one string is empty, the cost is just the length of the other string.
- If s_1 is empty, we would have to insert each character in the other string, starting from nothing.
- So edit(s_1[], s_2[1..j]) = *j (# of insertions)*
- edit(s_1[1..i], s_2[]) = *i (# of deletions)*

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \end{cases}$$

Computing Edit Distance

If the last characters are the same:

- edit(s_1[1..n-1], s_2[1..m-1])



If the last characters are different, three options:

- cost(delete) + edit(s_1[1..n-1], s_2[1..m]) → go up
- edit(s_1[1..n], s_2[1..m-1]) + cost(insert) → go left
- edit(s_1[1..n-1], s_2[1..m-1]) + cost(replace) → got left and up

We want the **minimum cost**, so if all costs are 1, our optimal substructure is

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1,j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1,j] + 1 \\ \text{Dist}[i,j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

Computing Edit Distance

Overlapping subproblems: Dist[i,j] = cost of operations to turn s_1[1...i] into s_2[1...j].

Optimal substructure:

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1,j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1,j] + 1 \\ \text{Dist}[i,j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ						
S						
I						
N						
G						
S						

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S						
I						
N						
G						
S						

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1					
I	2					
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1					
I	2					
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0				
I	2					
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0				
I	2					
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1			
I	2					
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1			
I	2					
N	3					
G	4					
S	5					

FIT2004: Lecture 6 - Dynamic Programming

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2		
I	2					
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	
I	2					
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2					
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1				
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1			
N	3					
G	4					
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1			
N	3					
G	4		Now	you	Try!	
S	5					

Example

$$\text{Dist}[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

i rows and j columns

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Outline

1. Introduction to Dynamic Programming
2. Coins Change
3. Unbounded Knapsack
4. 0/1 Knapsack
5. Edit Distance
6. Constructing Optimal Solution

Constructing optimal solutions

- The algorithms we have seen determine optimal values:
  - Minimum number of coins
  - Maximum value of knapsack
  - Edit distance
- How do we construct optimal solution that gives the optimal value:
  - The coins to give the change
  - The items to put in knapsack
  - Converting one string to the other
- There may be multiple optimal solutions and our goal is to return just one solution!
- Two strategies can be used.
  1. Create an additional array recording decision at each step
  2. Backtracking

Decision Array

- Make a second array of the same size.
- Each time you fill in a cell of the **memo array**, record your decision in the decision array.
- Remember, going right (or coming from the left) is insert.
- Going down (or coming from up) is delete.
- Going down and right (or coming from up and left) is replace OR do nothing.

	Φ	S	H	I	N	E
Φ	0					
S						
I						
N						
G						
S						

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S				
S						
I						
N						
G						
S						

	Φ	S	H	I	N	E
Φ	0	1				
S						
I						
N						
G						
S						

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H			
S						
I						
N						
G						
S						

	Φ	S	H	I	N	E
Φ	0	1	2			
S						
I						
N						
G						
S						

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I		
S						
I						
N						
G						
S						

	Φ	S	H	I	N	E
Φ	0	1	2	3		
S						
I						
N						
G						
S						

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	
S						
I						
N						
G						
S						

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	
S						
I						
N						
G						
S						

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S						
I						
N						
G						
S						

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S						
I						
N						
G						
S						

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S					
I	Delete I					
N	Delete N					
G	Delete G					
S	Delete S					

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1					
I	2					
N	3					
G	4					
S	5					

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing				
I	Delete I					
N	Delete N					
G	Delete G					
S	Delete S					

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0				
I	2					
N	3					
G	4					
S	5					

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H			
I	Delete I					
N	Delete N					
G	Delete G					
S	Delete S					

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1			
I	2					
N	3					
G	4					
S	5					

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I		
I	Delete I					
N	Delete N					
G	Delete G					
S	Delete S					

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2		
I	2					
N	3					
G	4					
S	5					

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	
I	Delete I					
N	Delete N					
G	Delete G					
S	Delete S					

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	
I	2					
N	3					
G	4					
S	5					

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I					
N	Delete N					
G	Delete G					
S	Delete S					

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2					
N	3					
G	4					
S	5					

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I				
N	Delete N					
G	Delete G					
S	Delete S					

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1				
N	3					
G	4					
S	5					

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I	Replace I,H	Do nothing	Insert N	Insert E
N	Delete N	Delete N	Replace N,H	Delete N	Do nothing	Insert E
G	Delete G	Delete G	Delete G	Delete G	Delete G	Replace G, E
S	Delete S	Delete S	Delete S	Delete S	Delete S	Delete S

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I	Replace I,H	Do nothing	Insert N	Insert E
N	Delete N	Delete N	Replace N,H	Delete N	Do nothing	Insert E
G	Delete G	Delete G	Delete G	Delete G	Delete G	Replace G, E
S	Delete S	Delete S	Delete S	Delete S	Delete S	Delete S

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

FIT2004: Lecture 6 - Dynamic Programming

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I	Replace I,H	Do nothing	Insert N	Insert E
N	Delete N	Delete N	Replace N,H	Delete N	Do nothing	Insert E
G	Delete G	Delete G	Delete G	Delete G	Delete G	Replace G, E
S	Delete S	Delete S	Delete S	Delete S	Delete S	Delete S

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I	Replace I,H	Do nothing	Insert N	Insert E
N	Delete N	Delete N	Replace N,H	Delete N	Do nothing	Insert E
G	Delete G	Delete G	Delete G	Delete G	Delete G	Replace G, E
S	Delete S	Delete S	Delete S	Delete S	Delete S	Delete S

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

FIT2004: Lecture 6 - Dynamic Programming

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I	Replace I,H	Do nothing	Insert N	Insert E
N	Delete N	Delete N	Replace N,H	Delete N	Do nothing	Insert E
G	Delete G	Delete G	Delete G	Delete G	Delete G	Replace G, E
S	Delete S	Delete S	Delete S	Delete S	Delete S	Delete S

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I	Replace I,H	Do nothing	Insert N	Insert E
N	Delete N	Delete N	Replace N,H	Delete N	Do nothing	Insert E
G	Delete G	Delete G	Delete G	Delete G	Delete G	Replace G, E
S	Delete S	Delete S	Delete S	Delete S	Delete S	Delete S

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I	Replace I,H	Do nothing	Insert N	Insert E
N	Delete N	Delete N	Replace N,H	Delete N	Do nothing	Insert E
G	Delete G	Delete G	Delete G	Delete G	Delete G	Replace G, E
S	Delete S	Delete S	Delete S	Delete S	Delete S	Delete S

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I	Replace I,H	Do nothing	Insert N	Insert E
N	Delete N	Delete N	Replace N,H	Delete N	Do nothing	Insert E
G	Delete G	Delete G	Delete G	Delete G	Delete G	Replace G, E
S	Delete S	Delete S	Delete S	Delete S	Delete S	Delete S

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

FIT2004: Lecture 6 - Dynamic Programming

Decision Array

	Φ	S	H	I	N	E
Φ	null	Insert S	Insert H	Insert I	Insert N	Insert E
S	Delete S	Do nothing	Insert H	Insert I	Insert N	Insert E
I	Delete I	Delete I	Replace I,H	Do nothing	Insert N	Insert E
N	Delete N	Delete N	Replace N,H	Delete N	Do nothing	Insert E
G	Delete G	Delete G	Delete G	Delete G	Delete G	Replace G, E
S	Delete S	Delete S	Delete S	Delete S	Delete S	Delete S

Sequence of operations:
Delete S (from position 5)
Replace G with E (at position 4)
Insert H (at position 2)

- SINGS
- SING
- SINE
- SHINE

Backtracking

- Start in bottom right.
- Are the letters the same?

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Start in bottom right.
- Are the letters the same?
- No

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Start in bottom right.
- Are the letters the same?
- No
- From recurrence…
- We know that our current value (3) was obtained from any of the three previous cells by adding 1.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Start in bottom right.
- Are the letters the same?
- No
- From recurrence…
- We know that our current value (3) was obtained from any of the three previous cells by adding 1.
- So our options are up or up-and-left.
- Choose one arbitrarily.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue from our new cell (but remember the path).

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue from our new cell (but remember the path).
- Letters are different.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue from our new cell (but remember the path).
- Letters are different.
- Must have come from the cell above.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Letters are the same.
- From our recurrence, we know:
  - If our value is the same as the up-and-left cell, then we could have came from there.
  - If our value is one more than the up cell or the left cell, then we could have come from there.
- In this case, we came from up-and-left.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Letter are the same.
- From our recurrence, we know:
  - If our value is the same as the up-and-left cell, then we could have came from there.
  - If our value is one more than the up cell or the left cell, then we could have come from there.
- In this case, we came from up-and-left.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.
- When you reach the top left cell, you are done.
- So the sequence of operations is:

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.
- When you reach the top left cell, you are done.
- So the sequence of operations is:
- Replace(5, E)

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.
- When you reach the top left cell, you are done.
- So the sequence of operations is:
- Replace(5, E), delete[4]

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.
- When you reach the top left cell, you are done.
- So the sequence of operations is:
- Replace(5, E), delete[4], nothing

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.
- When you reach the top left cell, you are done.
- So the sequence of operations is:
- Replace(5, E), delete[4], nothing, nothing

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.
- When you reach the top left cell, you are done.
- So the sequence of operations is:
- Replace(5, E), delete[4], nothing, nothing, insert(2, H)

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.
- When you reach the top left cell, you are done.
- So the sequence of operations is:
- Replace(5, E), delete[4], nothing, nothing, insert(2, H), nothing

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking

- Continue in this way.
- When you reach the top left cell, you are done.
- So the sequence of operations is:
- Replace(5, E), delete[4], nothing, nothing, insert(2, H), nothing
- SINGS
- SINGE (replace position 5 with E)
- SINE (delete G in position 4)
- SHINE (insert H at position 2)

	Φ	S	H	I	N	E
Φ	0	1	2	3	4	5
S	1	0	1	2	3	4
I	2	1	1	1	2	3
N	3	2	2	2	1	2
G	4	3	3	3	2	2
S	5	4	4	4	3	3

Backtracking Vs Decision array?

- Space usage:
  - Backtracking requires less space as it does not require creating an additional array.
  - However, space **complexity** is the same.
- Efficiency:
  - Backtracking requires to **determine** what decision was made which costs additional computation.
  - However, time **complexity** is the same.
- Note the simple space saving trick discussed can only be used when the solution does not need to be constructed.

Review yourselves: Subset Sum Problem

The **Subset Sum Problem** is a classic problem in computer science and combinatorics. Given a set of integers and a target sum, determine whether there is a subset of the given set whose sum is equal to the target.

It means, given a set of numbers N = $\{a_1, a_2, \ldots, a_n\}$ and a value V, is their a subset of N such that the sum of elements is exactly V?

Note: Unlike Coins Change problem, a number can only be used once to make the value V.

Example: Suppose N = {1, 5, 6, 9} and the value V is 13. The answer is FALSE because no subset of N adds to 13. For V=15, the answer is TRUE because 9 + 6 = 15.

It is an NP-complete problem, meaning no known polynomial-time algorithm exists for all inputs. It is a special case of the Knapsack Problem. It appears in many real-world problems such as resource allocation, cryptography, partitioning tasks, etc.

Concluding Remarks

Dynamic Programming Strategy

- Assume you already know the optimal solutions for all subproblems and have memoized these solutions.
- Observe how you can solve the original problem using this memoization.
- Iteratively solve the sub-problems and memoize.

Things to do (this list is not exhaustive)

- Practice, practice, practice:
  - http://www.geeksforgeeks.org/tag/dynamic-programming/
  - https://www.topcoder.com/community/data-science/data-science-tutorials/dynamic-programming-from-novice-to-advanced/

Coming Up Next

- Dynamic Programming Graph Algorithms.

Reading

- Course Notes: Chapter 7
- You can also check algorithms' textbooks for contents related to this lecture, e.g.:
  - CLRS: Chapter 15
  - KT: Chapter 6
  - Rou: Chapter 16