Dynamic Programming

Robin Visser

IOI Training Camp
University of Cape Town

3 December 2016

Overview

Background

- Dynamic programming is a programming technique which separates a problem into simpler sub-problems.

Background

- Dynamic programming is a programming technique which separates a problem into simpler sub-problems.
- Each sub-problem is calculated just once. When the same sub-problem is required to be calculated again, the stored solution is used instead of recomputing the sub-problem.

Background

- Dynamic programming is a programming technique which separates a problem into simpler sub-problems.
- Each sub-problem is calculated just once. When the same sub-problem is required to be calculated again, the stored solution is used instead of recomputing the sub-problem.
- It is a frequently used technique in competitions and can often reduce the time complexity of problems from exponential to polynomial.

Background

- Dynamic programming is a programming technique which separates a problem into simpler sub-problems.
- Each sub-problem is calculated just once. When the same sub-problem is required to be calculated again, the stored solution is used instead of recomputing the sub-problem.
- It is a frequently used technique in competitions and can often reduce the time complexity of problems from exponential to polynomial.

Example:

Background

- Dynamic programming is a programming technique which separates a problem into simpler sub-problems.
- Each sub-problem is calculated just once. When the same sub-problem is required to be calculated again, the stored solution is used instead of recomputing the sub-problem.
- It is a frequently used technique in competitions and can often reduce the time complexity of problems from exponential to polynomial.

Example: What is the value of $1 + 3 + 9 + 2 + 4 + 8 + 10$

Background

- Dynamic programming is a programming technique which separates a problem into simpler sub-problems.
- Each sub-problem is calculated just once. When the same sub-problem is required to be calculated again, the stored solution is used instead of recomputing the sub-problem.
- It is a frequently used technique in competitions and can often reduce the time complexity of problems from exponential to polynomial.

Example: What is the value of $1 + 3 + 9 + 2 + 4 + 8 + 10 + 1$

Fibonacci sequence

Problem

Calculate the nth Fibonacci number. (The Fibonacci sequence is generated as $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$

Fibonacci sequence

Problem

Calculate the nth Fibonacci number. (The Fibonacci sequence is generated as $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$

- One can easily code a recursive solution

Fibonacci sequence

Problem

Calculate the nth Fibonacci number. (The Fibonacci sequence is generated as $F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$

- One can easily code a recursive solution

```
def fibonacci(n):
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Fibonacci sequence

Problem

Calculate the nth Fibonacci number. (The Fibonacci sequence is generated as $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$

- One can easily code a recursive solution

```
def fibonacci(n):
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)
```

- This will take exponential time, therefore very slow! It would take about 4 trillion years to calculate F_{100} (longer than the age of the universe)

Fibonacci sequence

- Clearly, a better approach is required.

Fibonacci sequence

- Clearly, a better approach is required.
- Instead of recomputing the same values, we store them in memory. This is called *memoisation*.

Fibonacci sequence

- Clearly, a better approach is required.
- Instead of recomputing the same values, we store them in memory. This is called *memoisation*.
- If our result has been already computed, we simply retrieve the solution from memory instead of recomputing the result.

Fibonacci sequence

- Clearly, a better approach is required.
- Instead of recomputing the same values, we store them in memory. This is called *memoisation*.
- If our result has been already computed, we simply retrieve the solution from memory instead of recomputing the result.

```
def fibonacci(n):
    if memo[n] >= 0: return memo[n]
    if n <= 1: return n
    memo[n] = fibonacci(n-1) + fibonacci(n-2)
    return memo[n]
```

Fibonacci sequence

- This already optimises the problem down to linear time.

Fibonacci sequence

- This already optimises the problem down to linear time.
- We still require $\mathrm{O}(n)$ memory though.

Fibonacci sequence

- This already optimises the problem down to linear time.
- We still require $O(n)$ memory though.
- A *bottom-up* approach can reduce memory usage to constant space

Fibonacci sequence

- This already optimises the problem down to linear time.
- We still require O(n) memory though.
- A *bottom-up* approach can reduce memory usage to constant space

```
def fibonacci(n):
    if n == 0: return 0
    prevFib, curFib = 0, 1
    for i in range(n-1):
        newFib = prevFib + curFib
        prevFib, curFib = curFib, newFib
    return curFib
```

Fibonacci sequence

- This approach requires only O(n) time and O(1) memory.

Fibonacci sequence

- This approach requires only O(n) time and O(1) memory.
- Usually takes less time in practice due to function call overhead.

Fibonacci sequence

- This approach requires only O(n) time and O(1) memory.
- Usually takes less time in practice due to function call overhead.
- In general, there are three things to consider:

Fibonacci sequence

- This approach requires only $O(n)$ time and $O(1)$ memory.
- Usually takes less time in practice due to function call overhead.
- In general, there are three things to consider:
  - State space

Fibonacci sequence

- This approach requires only $O(n)$ time and $O(1)$ memory.
- Usually takes less time in practice due to function call overhead.
- In general, there are three things to consider:
  - State space
  - Recurrence relation

Fibonacci sequence

- This approach requires only $O(n)$ time and $O(1)$ memory.
- Usually takes less time in practice due to function call overhead.
- In general, there are three things to consider:
  - State space
  - Recurrence relation
  - Traversal

Fibonacci sequence

- This approach requires only $O(n)$ time and $O(1)$ memory.
- Usually takes less time in practice due to function call overhead.
- In general, there are three things to consider:
  - State space
  - Recurrence relation
  - Traversal
- Both approaches have their pros and cons. Recursion with memoisation can sometimes be easier to conceptualise (don't need to worry about traversal) although the fastest solutions can often only be done as a bottom-up DP.

Coin counting

Problem

Given a set of n coins, each with value $v_1, v_2, \ldots, v_n$, make change to the value of M using the least amount of coins

Coin counting

Problem

Given a set of n coins, each with value $v_1, v_2, \ldots, v_n$, make change to the value of M using the least amount of coins

- Let coins[x] be the optimal solution for making x change.

Coin counting

Problem

Given a set of n coins, each with value $v_1, v_2, \ldots, v_n$, make change to the value of M using the least amount of coins

- Let coins[x] be the optimal solution for making x change.
- Note that we having the following dependency:
  coins$[X] = 1 + \min\{$coins$[X - v_1, X - v_2, \ldots, X - v_i\}$
  for all i where $v_i \leq X$.

Coin counting

Problem

Given a set of n coins, each with value $v_1, v_2, \ldots, v_n$, make change to the value of M using the least amount of coins

- Let coins$[x]$ be the optimal solution for making x change.
- Note that we having the following dependency: coins$[X] = 1 + \min\{$coins$[X - v_1, X - v_2, \ldots, X - v_i\}$ for all i where $v_i \leq X$.
- This immediately suggests a DP approach.

Code

Pseudocode:

```
coins[0] = 0
for i from 1 to m:
  for j from 1 to n:
    if v[j] < i:
      coins[i] = min(coins[i], 1 + coins[i-v[j]])
return coins[m]
```

Code

Pseudocode:

```
coins[0] = 0
for i from 1 to m:
  for j from 1 to n:
    if v[j] < i:
      coins[i] = min(coins[i], 1 + coins[i-v[j]])
return coins[m]
```

- Notice that to calculate some value of coins[x] requires O(n) time.

Code

Pseudocode:

```
coins[0] = 0
for i from 1 to m:
  for j from 1 to n:
    if v[j] < i:
      coins[i] = min(coins[i], 1 + coins[i-v[j]])
return coins[m]
```

- Notice that to calculate some value of coins$[x]$ requires O(n) time.
- Final algorithm hences run in O(nM) time. (pseudo-polynomial time)

Code

Pseudocode:

```
coins[0] = 0
for i from 1 to m:
  for j from 1 to n:
    if v[j] < i:
      coins[i] = min(coins[i], 1 + coins[i-v[j]])
return coins[m]
```

- Notice that to calculate some value of coins[x] requires O(n) time.
- Final algorithm hences run in O(nM) time. (pseudo-polynomial time)
- This is a special case of the unbounded knapsack problem (where value of each object is 1)

Longest common subsequence

Problem

Given two strings, find the longest common subsequence.

Example: Longest common subsequence of **GAC** and **AGCAT** is {**AC**, **GC**, **GA**}.

Longest common subsequence

Problem

Given two strings, find the longest common subsequence.

Example: Longest common subsequence of **GAC** and **AGCAT** is {**AC**, **GC**, **GA**}.

- Can be done using a 2D dynamic programming approach.

Longest common subsequence

Problem

Given two strings, find the longest common subsequence.

Example: Longest common subsequence of **GAC** and **AGCAT** is {**AC**, **GC**, **GA**}.

- Can be done using a 2D dynamic programming approach.
- Consider the LCS of *prefixes* of the given strings.

Algorithm

- Given two strings X and Y, let X_i denote the first i character of X and Y_j denote the first j characters of Y.

Algorithm

- Given two strings X and Y, let X_i denote the first i character of X and Y_j denote the first j characters of Y.
- Let LCS$[i][j]$ denote the LCS of X_i and Y_j.

Algorithm

- Given two strings X and Y, let X_i denote the first i character of X and Y_j denote the first j characters of Y.
- Let LCS$[i][j]$ denote the LCS of X_i and Y_j.
- We have the following relation:

Algorithm

- Given two strings X and Y, let X_i denote the first i character of X and Y_j denote the first j characters of Y.
- Let LCS$[i][j]$ denote the LCS of X_i and Y_j.
- We have the following relation:

$$\text{LCS}[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \text{LCS}[i-1][j-1] + 1 & \text{if } x_i = y_j \\ \max(\text{LCS}[i][j-1], \text{LCS}[i-1][j]) & \text{if } x_i \neq y_j \end{cases}$$

Algorithm

Dynamic Programming

Robin Visser

- Given two strings X and Y, let X_i denote the first i character of X and Y_j denote the first j characters of Y.
- Let $\text{LCS}[i][j]$ denote the LCS of X_i and Y_j.
- We have the following relation:

$$\text{LCS}[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \text{LCS}[i-1][j-1] + 1 & \text{if } x_i = y_j \\ \max(\text{LCS}[i][j-1], \text{LCS}[i-1][j]) & \text{if } x_i \neq y_j \end{cases}$$

- Algorithm runs in $\text{O}(nm)$ time where n is length of X and m is length of Y.

Code

Pseudocode:

```
for i from 0 to m:    C[i][0] = 0
for j from 0 to n:    C[0][j] = 0
for i from 1 to m:
  for j from 1 to n:
    if X[i] = Y[j]:
      C[i][j] = C[i-1][j-1] + 1
    else:
      C[i,j] = max(C[i][j-1], C[i-1][j])
```

Code

Pseudocode:

```
for i from 0 to m:    C[i][0] = 0
for j from 0 to n:    C[0][j] = 0
for i from 1 to m:
  for j from 1 to n:
    if X[i] = Y[j]:
      C[i][j] = C[i-1][j-1] + 1
    else:
      C[i,j] = max(C[i][j-1], C[i-1][j])
```

- To recreate the subsequence, one can backtrack starting from C$[m][n]$.

Code

Pseudocode:

```
for i from 0 to m:    C[i][0] = 0
for j from 0 to n:    C[0][j] = 0
for i from 1 to m:
  for j from 1 to n:
    if X[i] = Y[j]:
      C[i][j] = C[i-1][j-1] + 1
    else:
      C[i,j] = max(C[i][j-1], C[i-1][j])
```

- To recreate the subsequence, one can backtrack starting from C[m][n].
- This is a commonly used technique in dynamic programming to recreate the optimal state required.

Subset sum

Problem

Given a set of n integers $x_1, x_2, \ldots, x_n$, determine if there exists a subset whose sum is S.

Subset sum

Problem

Given a set of n integers $x_1, x_2, \ldots, x_n$, determine if there exists a subset whose sum is S.

- Again, a 2D state space will be used.

Subset sum

Problem

Given a set of n integers $x_1, x_2, \ldots, x_n$, determine if there exists a subset whose sum is S.

- Again, a 2D state space will be used.
- We define a boolean valued function Q(i, s) to be true iff there is a nonempty subset of $x_1, \ldots, x_i$ which sums to s.

Algorithm

- Let A be the sum of the negative values and B the sum of the positive values.

Algorithm

- Let A be the sum of the negative values and B the sum of the positive values.
- We have the following relation:

Algorithm

- Let A be the sum of the negative values and B the sum of the positive values.
- We have the following relation:

$$\text{Q}[i][s] = \begin{cases} x_1 == s & \text{if } i = 1 \\ \textbf{false} & \text{if } s < A \text{ or } s > B \\ \text{Q}[i-1][s] \text{ or } x_i == s & \text{otherwise} \\ \quad \text{or } \text{Q}[i-1][s-x_i] & \end{cases}$$

Algorithm

- Let A be the sum of the negative values and B the sum of the positive values.
- We have the following relation:

$$\mathsf{Q}[i][s] = \begin{cases} x_1 == s & \text{if } i = 1 \\ \textbf{false} & \text{if } s < A \text{ or } s > B \\ \mathsf{Q}[i-1][s] \text{ or } x_i == s & \text{otherwise} \\ \text{ or } \mathsf{Q}[i-1][s-x_i] & \end{cases}$$

- Algorithm runs in $\mathsf{O}(n(B-A))$ time (pseudo-polynomial).

Code

Pseudocode:

```
Q[1][x1] = True
for i from 2 to n:
    for s from A to B:
        if Q[i-1][s] or Q[i-1][s-xi] or xi==s:
            Q[i][s] = True
return Q[n][S]
```

Code

Pseudocode:

```
Q[1][x1] = True
for i from 2 to n:
    for s from A to B:
        if Q[i-1][s] or Q[i-1][s-xi] or xi==s:
            Q[i][s] = True
return Q[n][S]
```

- To count number of subsets that sum to S, just replace boolean values with integer values and *add* instead of *or*.

Code

Pseudocode:

```
Q[1][x1] = True
for i from 2 to n:
    for s from A to B:
        if Q[i-1][s] or Q[i-1][s-xi] or xi==s:
            Q[i][s] = True
return Q[n][S]
```

- To count number of subsets that sum to S, just replace boolean values with integer values and *add* instead of *or*.
- Again, backtracking can be used to recreate the actual subset.

Summary

- Dynamic programming is a widely adaptable technique that can be used in many different situations.

Summary

- Dynamic programming is a widely adaptable technique that can be used in many different situations.
- Whenever different *states* exist and previous states can be used to construct bigger ones, it's probably DP.

Summary

- Dynamic programming is a widely adaptable technique that can be used in many different situations.
- Whenever different *states* exist and previous states can be used to construct bigger ones, it's probably DP.
- There can often be several different ways to do a DP with differing time complexities, so even if you have a valid solution, always try to find optimisations.