Dynamic Programming: Change Problem

Change problem

Input: An integer `money` and positive integers `coin[1], ..., coin[d]`

Output: The minimum number of coins with denominations `coin[1], ..., coin[d]` that changes money

Greedy algorithm picks the largest denomination available that does not exceed money.

```
GreedyChange(money):
Change <- empty collection of coins
while money > 0:
    coin <- largest denomination that does not exceed money
    add coin to Change
    money <- money - coin
return Change
```

Recursive algorithm picks the denominations which yields the minimum number of coins used. For example, there are 9 cents given denominations 6, 5, and 1. We can have 3 + 6, 4 + 5 or 8 + 1. Pick one that `money-coin[i]` requires the minimum number of coins used. In mathematics,

$$\text{MinNumCoins}(9) = \min \begin{cases} \text{MinNumCoins}(9-6)+1 \\ \text{MinNumCoins}(9-5)+1 \\ \text{MinNumCoins}(9-1)+1 \end{cases}$$

In general,

$$\text{MinNumCoins}(\text{money}) = \min \begin{cases} \text{MinNumCoins}(\text{money - coin}[1])+1 \\ \text{MinNumCoins}(\text{money - coin}[2])+1 \\ \dots \\ \text{MinNumCoins}(\text{money - coin}[d])+1 \end{cases}$$

```
RecursiveChange(money, coins):
if money = 0:
    return 0
MinNumCoins <- infinity
for i from 1 to len(coins):
    if money >= coin[i]:
        NumCoins <- RecursiveChange(money - coin[i], coins)
        if NumCoins + 1 < MinNumCoins:
            MinNumCoins <- NumCoins + 1
return MinNumCoins
```

Dynamic programming loops through each dollar. For each dollar, dynamic programming looks up the coins used by each possible coins, and records the best solution.

```
DPChange(money, coins):
MinNumCoins(0) <- 0
for m from 1 to money:
    MinNumCoins[m] <- infinity
    for i from 1 to len(coins):
        if m >= coin[i]:
            NumCoins <- MinNumCoins(m - coin[i]) + 1
            if NumCoins < MinNumCoins[m]:
                MinNumCoins[m] <- NumCoins
return MinNumCoins(money)
```

Dynamic Programming: String comparison

Alignment

An alignment of two strings is a two-row matrix:

1st row: symbols of the 1st string (in order) interspersed by "-"

2nd row: symbols of the 2nd string (in order) interspersed by "-"

Alignment score

Premium for every match ($+1$), and penalty for every mismatch ($-\mu$), indel ($-\sigma$).

Optimal alignment

Input: Two strings, mismatch penalty μ, and indel penalty σ.

Output: An alignment of the strings maximizing the score.

Longest common subsequence

Input: Two strings

Output: A longest common subsequence of these strings

Remark: It corresponds to maximizing the score of an alignment with $\mu = \sigma = 0$

Edit distance

Input: Two strings.

Output: The minimum number of operations (insertions, deletions, and substitutions of symbols) to transform one string into another.

Remark: Maximizing alignment score = maximizing edit distance

Suppose we want to transform A to B. Let $D(i, j)$ be the edit distance of an $i-$prefix $A[1...i]$ and a $j-$prefix $B[1...j]$.

$$D(i,j) = \min \begin{cases} D(i, j-1) + 1 \\ D(i-1, j) + 1 \\ D(i-1, j-1) + 1, A[i] \neq B[j] \\ D(i-1, j-1), A[i] = B[j] \end{cases}$$

```
EditDistance(A[1...n], B[1...m]):
D(i, 0) <- i and D(0, j) <- j for all i, j
for j from 1 to m:
    for i from 1 to n:
        insertion <- D(i, j - 1) + 1
        deletion <- D(i - 1, j) + 1
        match <- D(i - 1, j - 1)
        mismatch <- D(i - 1, j - 1) + 1
        if A[i] == B[j]:
            D(i, j) <- min(insertion, deletion, match)
        else:
            D(i, j) <- min(insertion, deletion, mismatch)
return D(n, m)
```

```
OutputAlignment(i, j):
if i == 0 and j == 0:
    return
if i > 0 and D(i, j) == D(i - 1, j) + 1:
    OutputAlignment(i - 1, j)
    print (A[i], '-')
else if j > 0 and D(i, j) == D(i, j - 1) + 1:
    OutputAlignment(i, j - 1)
    print('-', B[j])
else:
    OutputAlignment(i - 1, j - 1)
    print(A[i], B[j])
```

Dynamic Programming: Knapsack

Knapsack with repetitions problem

Input: Weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ of n items; total weight W (v_i's, w_i's, and W are non-negative integers)

Output: The maximum value of items whose weight does not exceed W. Each item can be used any number of times.

```
Knapsack(W):
value[0] <- 0
for w from 1 to W:
    value[w] <- 0
    for i from 1 to n:
        if w[i] <= w:
            val <- value[w - w[i]] + v[i]
            if val > value[w]:
                value[w] <- val
return value[W]
```

Subproblem

Let $\text{value}[w]$ be the maximum value of knapsack of weight w.

$$\text{value}[w] = \max_{i: w_i \leq w} \{\text{value}[w - w_i] + v_i\}$$

Explanation: For each possible capacity up to W, all items which can be put into the knapsack are considered. We need to compare the current value of knapsack `value[w]` and the optimal value of knapsack in previous step `value[w - w[i]]` plus the corresponding item's value `v[i]`.

Knapsack without repetitions problem

Input: Weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ of n items; total weight W (v_i's, w_i's, and W are non-negative integers)

Output: The maximum value of items whose weight does not exceed W. Each item can be used at most once.

```
Knapsack(W):
initialize all value[0, j] <- 0
initialize all value[w, 0] <- 0
for i from 1 to n:
    for w from 1 to W:
        value[w, i] <- value[w, i - 1]
        if w[i] <= w:
            val <- value[w - w[i], i - 1] + v[i]
            if value[w, i] < val:
                value[w, i] <- val
return value[W, n]
```

Subproblem

Let $\text{value}[w, i]$ be the maximum value of knapsack of weight w and items $1, \ldots, i$.

$$\text{value}[w, i] = \max\{\text{value}[w - w_i, i - 1] + v_i, \text{value}[w, i - 1]\}$$

Explanation: Compare the previous optimal solution with weight $w - w_i$ and the value of item i with the value of doing nothing $\text{value}[w, i - 1]$.

To back trace the past, we need to compare $\text{value}[w - w_i, i-1] + v_i$ and $\text{value}[w, i-1]$, starting from the optimal value.

Memoization

```
Knapsack(w):
if w is in hash table:
    return value[w]
value[w] <- 0
for i from 1 to n:
    if w[i] <= w:
        val <- Knapsack(v - w[i]) + v[i]
        if val > value[w]:
            value[w] <- val
insert value[w] into hash table with key w
return value[w]
```

The running time `O(nW)` is not polynomial since the input size is proportion to `log(W)`, but not `W`. In other words, the running time is `O(n2^(logW))` because the size of integer is represented in terms of `log(W)`.

https://stackoverflow.com/questions/4538581/why-is-the-knapsack-problem-pseudo-polynomial#answer-4538668

Dynamic Programming: Placing Parentheses

Example: How to place parentheses in an expression $1+2-3\times4-5$ to maximize its value?

$((((1+2)-3)\times4)-5)=-5$

$((1+2)-((3\times4)-5))=-4$

$((1+2)-(3\times(4-5)))=6$

Placing parentheses

Input: A sequence of digits $d_1, \ldots, d_n$ and a sequence of operations $op_1, \ldots, op_{n-1} \in \{+, -, \times\}$

Output: An order of applying these operations that maximizes the value of the expression $d_1\, op_1\, d_2\, op_2 \ldots op_{n-1}\, d_n$.

Subproblems

Let $E_{i,j}$ be the subexpression: $d_i\, op_i \ldots op_{j-1}\, d_j$.

Let $M(i,j)$ be the maximum value of $E_{i,j}$ and $m(i,j)$ be the maximum value of $E_{i,j}$

$$M(i,j) = \max_{i \le k \le j-1} \begin{cases} M(i,k)\ op_k\ M(k+1,j) \\ M(i,k)\ op_k\ m(k+1,j) \\ m(i,k)\ op_k\ M(k+1,j) \\ m(i,k)\ op_k\ m(k+1,j) \end{cases}$$

$$m(i,j) = \min_{i \le k \le j-1} \begin{cases} M(i,k)\ op_k\ M(k+1,j) \\ M(i,k)\ op_k\ m(k+1,j) \\ m(i,k)\ op_k\ M(k+1,j) \\ m(i,k)\ op_k\ m(k+1,j) \end{cases}$$

```
MinAndMax(i, j):
max <- Inf
min <- -Inf
for k from i to j - 1:
    a <- M(i, k) op[k] M(k + 1, j)
    b <- M(i, k) op[k] m(k + 1, j)
    c <- m(i, k) op[k] M(k + 1, j)
    d <- m(i, k) op[k] m(k + 1, j)
    min <- min(min, a, b, c ,d)
    max <- max(max, a, b, c, d)
return (min, max)
```

When computing $M(i,j)$, the values of $M(i,k)$ and $M(k+1,j)$ should already be computed. Solve all subproblems in order of increasing $j-i$.

```
Parentheses(d[1]op[1]d[2]op[2]...d[n]):
for i from 1 to n:
    m(i, i) <- d[i], M(i, i) <- d[i]
for s from 1 to n - 1:
    for i from 1 to n - s:
        j <- i + s
        m(i, j), M(i, j) <- MinAndMax(i, j)
return M(1, n)
```