Dynamic Programming

Data Structures and Algorithms (094224)

Tutorial 13

Winter 2022/23

Dynamic Programming

- Focus on optimization problems
  - Looking for a feasible solution that minimizes/maximizes some objective function
  - E.g., shortest (s, t)-path, minimum spanning tree
- An (optimal) solution to the problem can be constructed from (optimal) solutions to smaller subproblems
- A bottom-up approach
- Each subproblem is solved exactly once
- Solutions are stored in a lookup table
  - Accessed in the process of solving larger subproblems
- Trading space for time
  - Don't solve same subproblem many times, but solution has to be stored
- Define a recursive equation although dynamic programming algorithms are not recursive!

Question 1

Given an amount of money $M \in \mathbb{Z}_{>0}$ and n types of coin values $v(1) = 1 < v(2) < ... < v(n)$ (all integers), propose an algorithm that finds the minimal number of coins whose total value is equal to M.

Remark:

- Since there exists a coin with value 1 any amount of money M can be exchanged

Solution – first attempt

- Given M,n and v compute minimal number of coins by:
  1. Take as much coins as possible with the largest value, i.e., $\left\lfloor \frac{M}{v(n)} \right\rfloor$
  2. Calculate the remaining amount of money, i.e., $M = M - \left\lfloor \frac{M}{v(n)} \right\rfloor \cdot v(n)$
  3. Take as much coins with the second largest value, i.e., $\left\lfloor \frac{M}{v(n-1)} \right\rfloor$
  4. Calculate the remaining amount of money, i.e., $M = M - \left\lfloor \frac{M}{v(n-1)} \right\rfloor \cdot v(n-1)$
  5. Continue until $M = 0$
- Counter example:
- $M = 11, n = 4, v(1) = 1, v(2) = 5, v(3) = 6,\ v(4) = 9$
- The above procedure will return 3
  - Taking 1 coin with value 9 since $\left\lfloor \frac{11}{v(4)} \right\rfloor = 1$
  - Taking two coins with value 1 since $11 - \left\lfloor \frac{11}{v(4)} \right\rfloor = 2$ and $\left\lfloor \frac{2}{1} \right\rfloor = 2$
    - coins with value greater then the remaining amount are skipped
- The minimal number of coins is 2 since we can take one coin with value 5 and one with value 6

Solution

- Let $r(i)$ be the minimal number of coins with total value of i
  - $0 \leq i \leq M$
- Key observations: $r(0) = 0$ and

$$r(i) = \min_{1 \leq j \leq n,\, v(j) \leq i} \{1 + r(i - v(j))\}$$

  - Minimize exchange with coin j plus exchange the reminder, $(i - v(j))$, optimally
- $r(M)$ is the desired value

Solution — Pseudocode

Change_Money(M, V)

1: new array $r[0 \dots M]$
2: $r[0] = 0$
3: **for** $i = 1, \dots, M$ **do**
4: $\quad q = \infty$
5: $\quad$ **for** $j = 1, \dots, n$ **do**
6: $\quad\quad$ **if** $v[j] \leq i$ **then** $\triangleright$ if coin j's value is at most i
7: $\quad\quad\quad q = \min\{q, 1 + r[i - v[j]]\}$
8: $\quad r[i] = q$
9: return $r[M]$

Run time:

- $O(M)$ iterations loop in line 3
- $O(n)$ iterations loop in line 5
- $O(1)$ time for each inner–most iteration
- $O(nM)$ in total

Question 2

Given an unordered sequence of n numbers $M(1), \ldots, M(n)$, find the maximal number of elements that form a strictly increasing subsequence (not necessarily contiguous).

Example:
In the sequence $\langle 3, 2, 5, 4, 2, 3, 3, 4 \rangle$ the maximal number of element that form a strictly increasing subsequence is 3. An optimal solution is $\langle 3, \color{red}{2}, 5, 4, 2, 3, \color{red}{3}, \color{red}{4} \rangle$.

Solution

- Let $s(i)$ be the maximal length of a strictly increasing subsequence that can be formed from elements $M(1), \cdots, M(i)$ ending at position i
  - $1 \leq i \leq n$
- Key observations: $s(1) = 1$ and

$$s(i) = \begin{cases} \max\limits_{1 \leq j < i, M[j] < M[i]} \{1 + s(j)\}, & \exists j,\, 1 \leq j < i \; s.t \; M[j] < M[i] \\ 1, & otherwise \end{cases}$$

- $r = \max_{1 \leq i \leq n}\{s(i)\}$ is the desired value
  - The longest strictly increasing subsequence can end at any position

Solution — Pseudocode

```
Find_Max_Subsequence(M)
new array s[1 ... n]
s[1] = 1
for i = 2, ..., n do
    q = 1
    for j = 1, ..., i − 1 do
        if M[j] < M[i] then
            q = max{q, s[j] + 1}
    s[i] = q
max_val = 0
for i = 1, ..., n do
    if max_val < s[i] then
        max_val = s[i]
return max_val
```

Solution — cont.

Run time analysis

- $O(n)$ iterations of loop in line 3
- $O(n)$ iterations of loop in line 5
- $O(n)$ iterations of loop in line 10
- $O(1)$ time for each inner–most iteration
- $O(n^2 + n) = O(n^2)$ in total

Question 3 — Moed B Winter15-16

We are given n types of rectangular boxes. The dimensions width, length and height of box b denoted by $w(b)$, $\ell(b)$ and $h(b)$ respectively (real positive numbers). We would like to stack the boxes by placing box on top of a box. In order to maintain the stability of the stack a box b can be stacked on top of box b' only if $w(b) < w(b')$ and $\ell(b) < \ell(b')$, i.e., the base of each box in the stack must be strictly smaller (except the first box in the stack) than the box beneath her in the stack. The n types of boxes are characterized by n triples $\langle w_i, \ell_i, h_i \rangle$ such that for every box b of type $1 \leq i \leq n$ it holds $w(b) = w_i$, $\ell(b) = \ell_i$ and $h(b) = h_i$. Assume:

- There is an unlimited amount of each box type
- The n triples are sorted in a non-increasing width value, i.e., $w_1 \geq w_2 \geq \cdots \geq w_n$

Question 3 — Moed B Winter15-16 — cont.

1. Design an $O(n^2)$-time algorithm that calculates the height of the tallest stack possible with the restriction that the boxes cannot be rotated.
2. Design an $O(n^2)$-time algorithm that calculates the height of the tallest stack. Rotation of boxes is allowed.

Solution — 3.1 — Example

Consider the following boxes:

w_i	ℓ_i	h_i	box type
10	20	4	1
8	15	3	2
7	17	5	3
6	11	4	4
5	5	5	5

Solution — 3.1 — Example — cont.

- Can we stack box of type 1 on box of type 3?
  - NO! $w_1 \geq w_3$
- Is stacking $1 \leftarrow 2 \leftarrow 4 \leftarrow 5$ feasible?
  - Yes! $w_1 > w_2 > w_4 > w_5$ and $\ell_1 > \ell_2 > \ell_4 > \ell_5$
  - What is the height of stacking $1 \leftarrow 2 \leftarrow 4 \leftarrow 5$?
    - $h_1 + h_2 + h_4 + h_5 = 4 + 3 + 4 + 5 = 16$
- In an optimal solution must we use all types of boxes?
  - Optimal stacking $1 \leftarrow 3 \leftarrow 4 \leftarrow 5$ with height 18
- In an optimal solution at most one box of each type is used?
  - Yes! since no rotation is allowed once a box of type i is placed all boxes of type j on top must have dimensions $w_j < w_i$ and $\ell_j < \ell_i$

Solution — cont.

- Let $s(i)$ be the tallest stack of boxes with box i on top that can be formed with boxes of types $1, \cdots, i$
  - $1 \leq i \leq n$
- Denote by j an integer such that $1 \leq j < i$
- Key observations: $s(1) = h_1$ and

$$s(i) = \begin{cases} \max\limits_{j,(w_j > w_i) \wedge (\ell_j > \ell_i)} \{s(j)\} + h_i, & \exists j,\ s.t\,(w_j > w_i) \wedge (\ell_j > \ell_i) \\ h_i, & otherwise \end{cases}$$

- $r = \max_{1 \leq i \leq n}\{s(i)\}$ is the desired value
  - The tallest stack can end with any type of box
- Assume the input is given in an array A such that $A[i].w = w_i$, $A[i].\ell = \ell_i$ and $A[i].h = h_i$
- Notice, $\max\limits_{j,(w_j > w_i) \wedge (\ell_j > \ell_i)} \{s(j)\} + h_i = \max\limits_{j,(w_j > w_i) \wedge (\ell_j > \ell_i)} \{s(j) + h_i\}$

Solution — Pseudocode

`Tallest_Stack`(A)

```
new array s[1 ... n]
s[1] = A[1].h
for i = 2, ..., n do
    q = A[i].h
    for j = 1, ..., i − 1 do
        if A[j].w > A[i].w AND A[j].ℓ > A[i].ℓ then
            q = max{q, s[j] + A[i].h}
    s[i] = q
max_val = s[1]
for i = 2, ..., n do
    if max_val < s[i] then
        max_val = s[i]
return max_val
```

Solution — cont.

Run time analysis:

- $O(n)$ iterations of loop in line 3
- $O(n)$ iterations of loop in line 5
- $O(n)$ iterations of loop in line 10
- $O(1)$ time for each inner–most iteration
- $O(n^2 + n) = O(n^2)$ in total

Solution — 3.2 — Example

- We will solve using reduction to 3.1
- A box can be rotated in $3 \cdot 2 \cdot 1 = 6$ ways
  - Choose the width (3 options), choose the length (2 options) set the remaining value as height
- Consider the following box: $\langle 1, 2, 3 \rangle$
- The 6 orientations are $\langle 1, 2, 3 \rangle$, $\langle 1, 3, 2 \rangle$, $\langle 2, 1, 3 \rangle$, $\langle 2, 3, 1 \rangle$, $\langle 3, 1, 2 \rangle$, $\langle 3, 2, 1 \rangle$
- Once a box has been placed in the stack it is not possible to place the same type of box in the same orientation
  - Once a box i with dimensions $\langle w_i, \ell_i, h_i \rangle$ is placed all boxes on top must have dimensions $\langle w, \ell, h \rangle$ such that $w < w_i$ and $\ell < \ell_i$
- Since each orientation may be used at most once we can duplicate each type of box 6 times such that all orientations may be considered
- Since no rotation is needed we may use the solution of question 3.1

Solution — cont.

Algorithm:
Input: An array A of n types of boxes
output: The tallest stack possible

1. Create array B with all 6 orientations of all boxes
2. Sort (using $O(n \log n)$ sort) array B with respect to width
3. Run $\texttt{Tallest\_Stack}(B)$ and return its output

Run time analysis:

- Line 1 – $O(n)$
- Line 2 – $O(n \log n)$
- Line 3 – $O(n^2)$
- In total – $O(n^2)$

Question 4

We are given an array S of n symbols 'True', 'False', and an array O of $n-1$ binary operators 'and', 'or', and 'xor'. An expression from arrays S and O is $S[1]O[1]S[2]O[2]\cdots S[n-1]O[n-1]S[n]$.
Count the number of ways to place parentheses in the expression such that its value will evaluate to True.

Solution

AND	*True*	*False*
True	*True*	*False*
False	*False*	*False*

OR	*True*	*False*
True	*True*	*True*
False	*True*	*False*

XOR	*True*	*False*
True	*False*	*True*
False	*True*	*False*

Solution — cont.

- Example: $S = (True, True, False)$, $O = (xor, and)$. The expression is $True\ xor\ True\ and\ False$.
  Only one way to place parentheses such that the expression evaluates to True: $(True\ xor\ (True\ and\ False))$
- Recall the Matrix chain multiplication from lecture
- Let $T(i, j)$ be the number of ways to parenthesize the subexpression $S[i]O[i] \cdots O[j-1]S[j]$ such that its value is $True$
  - $1 \leq i \leq j \leq n$
  - In the above example $T(1,3) = 1$, $T(1,2) = T(2,3) = 0$
- $T(1, n)$ is the desired value
- For all $1 \leq i \leq n$

$$T(i,i) = \begin{cases} 1, & S[i] = True \\ 0, & S[i] = False \end{cases}$$

Solution — cont.

Attempt 1:

- Solution to some subexpression, $T(i,j)$, consist of splitting the subexpression at all $i \leq k < j$ and summing
- Is $T(i,j) = \sum_{k=i}^{j-1} T(i,k)T(k+1,j)$?
    - Consider the above example: is $T(1,3) = T(1,1)T(2,3) + T(1,2)T(3,3)$?
    - No, since $T(2,3) = 0$ and $T(1,2) = 0$ it follows $T(1,3) = 0$ but the parenthesized expression: (*True xor* (*True and False*)), evaluates to *True*
    - We must also count the number of ways to parenthesize subexpressions such that their value is *False*

Correct solution:

- Let $F(i,j)$ be the number of ways to parenthesize subexpression $S[i]O[i]\cdots O[j-1]S[j]$ such that its value is *False*
- For all $1 \leq i \leq n$, $F(i,i) = 1 - T(i,i)$

Solution — cont.

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k)T(k+1,j), & O[k] = and \\ T(i,k)T(k+1,j) + T(i,k)F(k+1,j) + F(i,k)T(k+1,j), & O[k] = or \\ T(i,k)F(k+1,j) + F(i,k)T(k+1,j), & O[k] = xor \end{cases}$$

$$F(i,j) = \sum_{k=i}^{j-1} \begin{cases} F(i,k)T(k+1,j) + F(i,k)F(k+1,j) + T(i,k)F(k+1,j), & O[k] = and \\ F(i,k)F(k+1,j), & O[k] = or \\ T(i,k)T(k+1,j) + F(i,k)F(k+1,j), & O[k] = xor \end{cases}$$

Solution – Pseudocode

Count_Par_True_Init(S)

```
new table T[1...n, 1...n]
new table F[1...n, 1...n]
for i = 1,...,n do
    for j = 1,...n do
        if i == j then
            if S[i] == True then
                T[i,i] = 1
                F[i,i] = 0
            else
                F[i,i] = 1
                T[i,i] = 0
        else
            T[i,j] = 0
            F[i,j] = 0
return (T, F)
```

Solution — Pseudocode

Count_Par_True(S, O)

```
(T, F) = Count_Par_True_Init(S)
for l = 2, ..., n do
    for i = 1, ..., n − l + 1 do
        j = i + l − 1
        for k = i, ..., j − 1 do
            if O[k] == and then
                T[i, j] = T[i, j] + T[i, k]T[k + 1, j]
                F[i, j] = F[i, j] + F[i, k]T[k + 1, j] + F[i, k]F[k + 1, j] + T[i, k]F[k + 1, j]
            else
                if O[k] == or then
                    T[i, j] = T[i, j] + T[i, k]T[k + 1, j] + T[i, k]F[k + 1, j] + F[i, k]T[k + 1, j]
                    F[i, j] = F[i, j] + F[i, k]F[k + 1, j]
                else
                    T[i, j] = T[i, j] + T[i, k]F[k + 1, j] + F[i, k]T[k + 1, j]
                    F[i, j] = F[i, j] + T[i, k]T[k + 1, j] + F[i, k]F[k + 1, j]
return T[1, n]
```

Solution — cont.

Run time analysis:

- Initialization takes $O(n^2)$ time
- 3 nested loops, each with $O(n)$ iterations
- $O(1)$ time for each inner–most iteration
- $O(n^3)$ time in total