COMP SCI 577 Homework 04 Problem 3

Dynamic Programming

Ruixuan Tu

rtu7@wisc.edu

University of Wisconsin-Madison

11 October 2022

Algorithm

Explanation

Denote input be k layers, n substances, and $e(i, j)$ for energy produced between the pair (i, j) of 2 substances for all $0 \leq i \leq j < n$.

For question (a), we want to calculate $One(l, r)$ for every right half-open interval $[l, r)$ such that $One(l, r)$ is the minimum energy for the substances consisting of substances s_l through s_r for all $0 \leq l \leq r < n$ and $k = 1$. We observe that with the base case $One(l, l) = 0$, there is a recursion $One(l, r) = One(l, r-1) + \sum_{m=l}^{r-1} e(m, r-1)$. The sum part is $O(n)$ for a trivial solution, but we could further reduce the complexity to $O(1)$ by prefix sum subroutine `getPrefixSum(iStart, iEnd, j)` for $\sum_{i=i_{\text{start}}}^{i_{\text{end}}} e(i, j) = e_{\text{prefix}}(i_{\text{end}}, j) - e_{\text{prefix}}(i_{\text{start}}, j)$. For the prefix sum, we need a preprocessing subroutine `preparePrefixSum(left, right)` to calculate $e_{\text{prefix}}(i, j) = \sum_{k=0}^{i} e(k, j) = e_{\text{prefix}}(i-1, j) + e(i, j)$ with base case $e_{\text{prefix}}(i, i) = e(i, i) = 0$ for every $0 \leq i \leq j < n$ with a complexity of $O(n^2)$, so $[i, j]$ is a closed interval for e_{prefix} that we need to call with $i_{\text{start}} = l - 1, i_{\text{end}} = r - 1, j = r - 1$ to get the correct sum converted from the right half-open interval $[l, r)$. The calculation is done in the subroutine `solveFirstLayer(left, right)` with the left bound not moving for the constraint in the recursion, so we should call this subroutine n times to calculate for all possible left bounds.

For question (b), we want to calculate $OPT(l, k) = \min_{l \leq m \leq r}(One(l, m) + OPT(m, k-1))$ for every right half-open interval $[l, k)$ such that $OPT(l, k)$ is the minimum energy for the substances s_l through s_{n-1}, with the base case $OPT(l, 1) = One(l, n)$, as the first layer is already calculated in question (a). The equation is of two parts: the left divided part $[l, m)$ with no further division so the first layer is reached to use $One(l, m)$, as well as the right divided part

$[m,n)$ with still $k-1$ layers to divide into to use $OPT(m,k-1)$, for every divider m to split the interval $[l,n)$ to $[l,m)$ and $[m,n)$. The calculation is done in the subroutine `solve(left, right, layers)` with the right bound not moving which is not necessary for generating all permutations. Thus, the result is $OPT(0,k)$ for the largest interval.

For history versions, version 1 uses the trivial $O(n^3)$ version of the sum to calculate the first layer $OPT(l,r,1)$, and version 1 and 2 uses the $O(kn^3)$ to calculate further $OPT(l,r,k) = \min_{l\leq m\leq r}(OPT(l,m,1)+OPT(m,r,k-1))$ which is replaced by the $O(n^2)$ calculation, as $OPT(l,m,k-1)$ is never used but causes a extra layer of calculation of $O(n)$.

Code (Python) of $O(n^2)+O(kn^2)$, Iterative (Version 3, Final)

```python
import numpy as np
import input

def preparePrefixSum(left: int, right: int) -> None: # O(n^2) for prefix sum preprocessing
    for j in range(left, right, 1):
        for i in range(left, right, 1):
            if i == 0:
                e_prefix[i][j] = e[i][j]
            e_prefix[i][j] = e_prefix[i - 1][j] + e[i][j]

def getPrefixSum(iStart: int, iEnd: int, j: int) -> int:
    if iStart > iEnd:
        raise Exception("invalid interval")
    low: int = 0 if iStart < 0 else e_prefix[iStart][j]
    high: int = 0 if iEnd < 0 else e_prefix[iEnd][j]
    return high - low

def solveFirstLayer(left: int, right: int) -> int: # O(n), [left, right), left does not move
    if left >= right - 1:
        return 0
    if one[left][right] != 0:
        return one[left][right]
```

```python
    col_sum: int = getPrefixSum(left - 1, right - 1, right - 1) # O(1) for prefix sum
    one[left][right] = solveFirstLayer(left, right - 1) + col_sum
    return one[left][right]

def solve(left: int, right: int, layers: int) -> int: # O(n^2*k): k layers * n^2 ways to choose intervals per layer
    for l in range(left, right, 1):
        dp[l][1] = one[l][right]
    for k in range(2, layers + 1, 1):
        for l in range(left, right, 1):
            candidates = []
            for m in range(l, right + 1, 1): # divider: left [i, m), right [m, right)
                candidates.append(one[l][m] + dp[m][k - 1])
            dp[l][k] = 0 if len(candidates) == 0 else min(candidates)
    return dp[left][layers]

if __name__ == "__main__":
    k: int = input.nextInt()
    n: int = input.nextInt()
    e = np.zeros((n, n), dtype=int)
    e_prefix = np.zeros((n, n), dtype=int)
    one = np.zeros((n, n + 1), dtype=int)
    dp = np.zeros((n + 1, k + 1), dtype=int)
    for i in range(n):
        for j in range(n - i - 1):
            e[i][i + j + 1] = input.nextInt()
    preparePrefixSum(0, n)
    for i in range(n): # O(n^2)
        solveFirstLayer(i, n) # O(n)
    print("sol={}".format(solve(0, n, k))) # O(n^2*k)
    for i in range(1, k + 1, 1):
        print("k={}".format(i))
```

```
        print(dp[:, i])
```

Proof

Question (a)

Induction

Claim: `solveFirstLayer(left, right)` calculate all $One(l,m)$ with $l < m \leq r$ for all $r-l \in \mathbb{N}$ and $l,r,m \in \mathbb{N}$.

Base Case: $r-l=0$ i.e. $One(l,l)=0$, the energy $One(l,l)$ and $e(l,l)$ must be 0, as a substance cannot react with itself.

Inductive Step: Suppose the sub-problem $One(l,r-1)$ is correct, and we want to prove that $One(l,r)$ is correct. From the induction hypothesis, we already have $One(l,r-1) = e(l,l) + (e(l,l+1)+e(l+1,l+1)) + [e(l,l+2)+e(l+1,l+2)+e(l+2,l+2)] + \cdots + [e(l,r-1)+e(l+1,r-1)+\cdots+e(r-1,r-1)]$, so that for $One(l,r)$ we still need to add $e(l,r)+e(l+1,r)+\cdots+e(r,r)$ which is the right part of the equation without transformation from right half-open interval to closed interval for $e(l,r)$. Therefore, the equation $One(l,r) = One(l,r-1) + \sum_{m=l}^{r-1} e(m,r-1)$ holds.

Termination

The functions `preparePrefixSum(left, right)` and `getPrefixSum(iStart, iEnd, j)` are iterative that must be terminated. The only recursive function `solveFirstLayer(left, right)` is terminated correctly, as it will return 0 for $r-l \leq 0$ as there is nothing to calculate on or further than the base case, and $r-l$ is decreasing on the recursion tree to reduce to the base case.

Complexity

As described in Explanation, `solveFirstLayer(left, right)` calculate all $One(l,m)$ with $l < m \leq n$. We should run it $n = O(n)$ times to calculate for $0 \leq l < n$. A call to `solveFirstLayer(left, right)` costs $O(n)$, as there is 1 recursion which decreases by 1 every time with $r-l = O(n)$ steps, and the prefix sum costs $O(1)$ to get the value and $O(n^2)$ to preprocess. Then the complexity of `solveFirstLayer(left, right)` is $O(n^2)$, and the complexity of `preparePrefixSum(left, right)` is $O(n^2)$. Details of the prefix sum are described in Explanation. Therefore, the overall complexity is $O(n^2)$.

Question (b)

Induction

Claim: This algorithm is correct for any layer $k \in \mathbb{N}^+$.
Base Case: $k = 1$ i.e. there is no further division, which is proved in Question (a), and $OPT(l,1) = One(l,n)$ is adherent to the definition, as described in Explanation.
Inductive Step: Suppose the sub-problem $OPT(l,k-1)$ is correct, and we want to prove that $OPT(l,k)$ is correct. As described in Explanation, $OPT(l,k) = \min_{l \le m \le r}(One(l,m) + OPT(m,k-1))$ for every right half-open interval $[l,k)$ such that $OPT(l,k)$ is the minimum energy for the substances s_l through s_{n-1}. This iteration includes all possibilities for intervals starting with all m such that $l \le m \le n$ and ending with n. We do not want the right bound to be moved as we only depend on different left bounds in finding the optimal solution.

Termination

The function `solve(left, right, layers)` is iterative that must be terminated.

Complexity

The function `solve(left, right, layers)` is iterative with 4 loops: a non-nested for $L \le l < R$ (base case, L and R are bounds) and a 3-nested for $2 \le k \le K, L \le l < R, l \le m \le R$ (inductive step, K is total number of bottles). The complexity of the non-nested loop is $O(n)$ trivially, and the complexity of the nested loop is $O(kn^2)$ for the k layers and $O(n^2)$ intervals for the left part without division bounded by $[l,m)$.

Test Cases

Input: $k \leftarrow 2$; $n \leftarrow 3$; $e_{1,2} \leftarrow 10$, $e_{1,3} \leftarrow 5$, $e_{2,3} \leftarrow 42$;
Output: 10;
Explanation: Same as the example in the write-out of this problem.

Input: $k \leftarrow 3$; $n \leftarrow 4$; $e_{i,j} \leftarrow \begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \\ & & \end{bmatrix}$;
Output: 1;
Explanation: 3 bottles, 4 substances: we have at most 2 substances in 1 bottle, resulting in minimum total energy of 1.

Input: $k \leftarrow 4$; $n \leftarrow 6$; $e_{i,j} \leftarrow \begin{bmatrix} 5 & 5 & 5 & 5 & 5 \\ & 5 & 5 & 5 & 5 \\ & & 1000 & 5 & 5 \\ & & & 1 & 1 \\ & & & & 1 \\ & & & & \end{bmatrix}$;

Output: 1;

Explanation: 4 bottles, 6 substances $s_0, \ldots s_5$: we have the optimal solution as $\{s_0\}\{s_1\}\{s_2\}\{s_3, s_4, s_5\}$ with minimum total energy of $e_{3,4} + e_{3,5} + e_{4,5} = 3$.

History Versions

Code (Python) of $O(n^2) + O(kn^3)$, Iterative (Version 2)

```python
import numpy as np
import input

def preparePrefixSum(left: int, right: int) -> None: # O(n^2) for prefix sum preprocessing
    for j in range(left, right, 1):
        for i in range(left, right, 1):
            if i == 0:
                e_prefix[i][j] = e[i][j]
            e_prefix[i][j] = e_prefix[i - 1][j] + e[i][j]

def getPrefixSum(iStart: int, iEnd: int, j: int) -> int:
    if iStart > iEnd:
        raise Exception("invalid interval")
    low: int = 0 if iStart < 0 else e_prefix[iStart][j]
    high: int = 0 if iEnd < 0 else e_prefix[iEnd][j]
    return high - low

def solveFirstLayer(left: int, right: int) -> int: # O(n), [left, right), left does not move
    if left >= right - 1:
```

```python
        return 0
    if dp[left][right][1] != 0:
        return dp[left][right][1]
    col_sum: int = getPrefixSum(left - 1, right - 1, right - 1) # O(1) for prefix sum
    dp[left][right][1] = solveFirstLayer(left, right - 1) + col_sum
    return dp[left][right][1]

def solve(left: int, right: int, layers: int) -> int: # O(n^3*k) k layers + n^2 ways to choose intervals per layer, [left, right)
    for layer in range(2, layers + 1, 1):
        for i in range(left, right, 1):
            for j in range(i, right + 1, 1):
                candidates = []
                for k in range(i, j, 1): # divider, left [i, k), right [k, j)
                    candidates.append(dp[i][k][1] + dp[k][j][layer - 1])
                dp[i][j][layer] = 0 if len(candidates) == 0 else min(candidates)
    return dp[left][right][layers]

if __name__ == "__main__":
    k: int = input.nextInt()
    n: int = input.nextInt()
    e = np.zeros((n, n), dtype=int)
    e_prefix = np.zeros((n, n), dtype=int)
    dp = np.zeros((n, n + 1, k + 1), dtype=int)
    for i in range(n):
        for j in range(n - i - 1):
            e[i][i + j + 1] = input.nextInt()
    preparePrefixSum(0, n)
    for i in range(n): # O(n^2)
        solveFirstLayer(i, n) # O(n)
```

```python
    print("sol={}".format(solve(0, n, k))) # O(n^3*k)
    for i in range(1, k + 1, 1):
        print("k={}".format(i))
        print(dp[:, :, i])
```

Code (Python) of $O(n^3) + O(kn^3)$, Recursive (Version 1)

```python
import numpy as np
import input

def solveFirstLayer(left: int, right: int) -> int: # O(n^2), [left, right)
    if left >= right - 1:
        return 0
    if dp[left][right][1] != 0:
        return dp[left][right][1]
    col_sum: int = 0
    for i in range(left, right, 1): # O(n): O(1) by prefix sum with independent preprocessing O(n^2)
        col_sum += e[i][right - 1]
    dp[left][right][1] = solveFirstLayer(left, right - 1) + col_sum
    return dp[left][right][1]

def solve(left: int, right: int, k: int) -> int: # O(n^3*k): k layers * O(n^2) ways to choose intervals per layer [left, right) * O(n) local work
    if left >= right - 1:
        return 0
    if k == 1:
        return dp[left][right][1]
    if dp[left][right][k] != 0:
        return dp[left][right][k]
    candidates = []
    for divider in range(left, right + 1, 1): # O(n)
```

```python
            candidates.append(solve(left, divider, 1) + solve(divider, right, k - 1))
    dp[left][right][k] = min(candidates)
    return dp[left][right][k]

if __name__ == "__main__":
    k: int = input.nextInt()
    n: int = input.nextInt()
    e = np.zeros((n, n), dtype=int)
    dp = np.zeros((n, n + 1, k + 1), dtype=int)
    for i in range(n):
        for j in range(n - i - 1):
            e[i][i + j + 1] = input.nextInt()
    for i in range(n): # O(n^3)
        solveFirstLayer(i, n) # O(n^2)
    print("sol={}".format(solve(0, n, k))) # O(n^3*k)
    for i in range(1, k + 1, 1):
        print("k={}".format(i))
        print(dp[:, :, i])
```

Code (Python) for utility `input`

```python
from typing import List, Optional

file = None
queue = []

def openFile(filename: Optional[str]):
    global file
    if filename != None:
        file = open(filename, "r")
    else:
        file = None

def next() -> Optional[str]:
```

```
    while len(queue) == 0:
        if file == None:
            line: str = input()
        else:
            line: str = file.readline()
        if len(line) == 0:
            return None
        lineArr: List[str] = line.split(" ")
        for lineElem in lineArr:
            queue.append(lineElem)
    return queue.pop(0)

def nextInt() -> int:
    result = next()
    if result != None:
        return int(result)
    raise Exception("no input")
```