106119029 , Algos Lab 8(Matrix Chain Multiplication)

Dipesh Kafle

Question 1

- Given a sequence ofmatrices such that any matrix may be multiplied by the previous matrix, write a program to find the best association such that the result is obtained with the minimum number of arithmetic operations.

Code

```cpp
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <numeric>
#include <sstream>
#include <string>
#include <string_view>
#include <unordered_map>
#include <vector>

using namespace std;

#define INF INT32_MAX

void solve_rec(vector<vector<int>> &m, vector<vector<int>> &s,
               vector<int> &mats, int i, int j) {
  if (m[i][j] != INF) {
    return;
  }
  for (int k = i; k < j; k++) {
    solve_rec(m, s, mats, i, k);
    solve_rec(m, s, mats, k + 1, j);
    if (m[i][j] > m[i][k] + m[k + 1][j] + mats[i - 1] * mats[k] * mats[j]) {
      s[i][j] = k;
      m[i][j] = m[i][k] + m[k + 1][j] + mats[i - 1] * mats[k] * mats[j];
    }
  }
}
```

```cpp

void solve_non_rec(vector<vector<int>> &m, vector<vector<int>> &s,
                   vector<int> &mats, int n) {
  for (int len = 2; len <= n; len++) { // len is length of chain
    for (int i = 1; i <= n - len + 1;
         i++) { // take all possible l length matrix sets
      int j = i + len - 1;
      for (int k = i; k <= j - 1; k++) {
        int cost = m[i][k] + m[k + 1][j] + mats[i - 1] * mats[k] * mats[j];
        if (cost < m[i][j]) {
          m[i][j] = cost;
          s[i][j] = k;
        }
      }
    }
  }
}

void print_parens(vector<vector<int>> &s, int i, int j) {
  if (i == j) {
    std::cout << "A" << i;
  } else {
    std::cout << "(";
    print_parens(s, i, s[i][j]);
    print_parens(s, s[i][j] + 1, j);
    std::cout << ")";
  }
}

string print_elem(int elem) { return elem == INF ? "INF" : to_string(elem); }

```

```cpp
int main() {
  for (auto mats : vector<vector<int>>{{77, 43, 26},
                                       {46, 32, 55, 53, 35},
                                       {62, 47, 79, 29, 81, 23, 70},
                                       {62, 47, 79, 45, 65, 65, 43, 45}}) {
    std::cout << "FOR dimensions: ";
    for (auto x : mats) {
      std::cout << x << " ";
    }
    std::cout << '\n';
    int n = mats.size() - 1;
    vector<vector<int>> m(mats.size(), vector<int>(mats.size(), INF));
    vector<vector<int>> s(mats.size(), vector<int>(mats.size(), INF));
    for (int i = 0; i <= n; i++) {
      m[i][i] = 0;
    }
    solve_rec(m, s, mats, 1, n);
    std::cout << m[1].back() << '\n';
    print_parens(s, 1, n);
    std::cout << '\n';
    std::cout << "Resultant Matrix m: \n";
    for (auto &rows : m) {
      for (auto elem : rows) {
        std::cout << std::left << std::setw(8) << print_elem(elem) << ' ';
      }
      std::cout << '\n';
    }
```

```cpp
    std::cout << "Resultant Matrix s: \n";
    for (auto &rows : s) {
      for (auto elem : rows) {
        std::cout << std::left << std::setw(8) << print_elem(elem) << ' ';
      }
      std::cout << '\n';
    }
    std::cout << '\n' << '\n';
  }
}
```

Question 2

- Which programming paradigm will be useful for you and why?

  Dynamic Programming is the programming paradigm that will be useful to solve this problem.

  if we have four matrices ABCD, we compute the cost required to find each of (A)(BCD), (AB)(CD), and (ABC)(D), making recursive calls to find the minimum cost to compute ABC, AB, CD, and BCD. We then choose the best one.

  However, this algorithm has exponential runtime complexity making it as inefficient as the naive approach of trying all permutations. The reason is that the algorithm does a lot of redundant work. For example, above we made a recursive call to find the best cost for computing both ABC and AB. But finding the best cost for computing ABC also requires finding the best cost for AB. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs.

  One simple solution is called memoization: each time we compute the minimum cost needed to multiply out a specific subsequence, we save it. If we are ever asked to compute it again, we simply give the saved answer, and do not recompute it. Since there are about $n^2/2$ different subsequences, where n is the number of matrices, the space required to do this is reasonable. It can be shown that this simple trick brings the runtime down to O(n^3) from O(2^n), which is more than efficient enough for real applications. This is top-down dynamic programming.

Question 3

- Show the stepwise execution(e.g. tabular structure)of your programfor three different set of inputs (varying both the number of matrices in the chain and the dimension of the matrices).

Output

```
FOR dimensions: 77 43 26
86086
(A1A2)
Resultant Matrix m:
0       INF     INF
INF     0       86086
INF     INF     0
Resultant Matrix s:
INF     INF     INF
INF     INF     1
INF     INF     INF


FOR dimensions: 46 32 55 53 35
204160
(A1((A2A3)A4))
Resultant Matrix m:
0       INF     INF     INF     INF
INF     0       80960   171296  204160
INF     INF     0       93280   152640
INF     INF     INF     0       102025
INF     INF     INF     INF     0
Resultant Matrix s:
INF     INF     INF     INF     INF
INF     INF     1       1       1
INF     INF     INF     2       3
INF     INF     INF     INF     3
INF     INF     INF     INF     INF
```

```
FOR dimensions: 62 47 79 29 81 23 70
358961
((A1(A2(A3(A4A5))))A6)
Resultant Matrix m:
0       INF     INF     INF     INF     INF     INF
INF     0       230206  192183  337821  259141  358961
INF     INF     0       107677  218080  192119  267789
INF     INF     INF     0       185571  106720  233910
INF     INF     INF     INF     0       54027   100717
INF     INF     INF     INF     INF     0       130410
INF     INF     INF     INF     INF     INF     0
Resultant Matrix s:
INF     INF     INF     INF     INF     INF     INF
INF     INF     1       1       3       1       5
INF     INF     INF     2       3       2       5
INF     INF     INF     INF     3       3       5
INF     INF     INF     INF     INF     4       5
INF     INF     INF     INF     INF     INF     5
INF     INF     INF     INF     INF     INF     INF
```

Question 4

- Find the worst case time and space complexity of your program and tally theoretically.

  Space Complexity for this dp approach is pretty straight forward n^2 , as we only need a matrix to store the computations.

  Time complexity for this approach is O(n^3). This isnt that obvious from the top down dp approach but when we find out the dependencies and make it a bottom up algorithm, it is obvious. The bottom up approach is done in `solve_non_rec` function in my code. One way to reason about this is , we are trying to fill a N X N matrix and to get the value of each of the cell, we must do O(N) work(calculation of the minimum multiplication costs from all possible splits). To find out m[i][j], we must find the cost of splitting (i,j) at all k, $i<k<j$ and take the minimum of those. Hence we can say it is a O(N^3) algorithm.