P4 報告 (延期一周繳交)

1. Title and Author

Assignment number - P4
Name - 潘煒翔
Student number - 112950009
Email - panweikyle.sc12@nycu.edu.tw

---

2. Description of the Problem

實作一個 **Sparse Matrix（稀疏矩陣）結構**，
能在不展開成完整二維陣列的情況下進行加法與乘法運算

列點要求：

- 每一列以「**鏈結串列 (linked list)**」形式儲存非零元素
  運算時間需達到：
- 加法：$O(m + e)$，其中 e 為非零元素數
- 乘法：$O(ln + e)$
- 不可展開成一般 m×n 陣列
- 提供輸入、輸出、記憶體單元素統計功能

並提供兩種操作模式：

1. **Manual Input Mode (a)**
   手動輸入兩個矩陣，並印出加法與乘法結果
2. **Random Generation Mode (b)**
   自動生成稀疏/密集矩陣，測量運算時間

---

3. Design Overview（設計總覽）

目標

1. 輸入兩個矩陣
2. 檢查是否可相加/相乘

3. 印出結果（dense 與 list+memory）；
   或改用隨機模式自動產生測資並進行效能測試。

程式選單

1. **Manual input (a)**
   - 輸入 A 維度（`4 5` 或 `(4,5)` 均可）
   - 逐列輸入 A 的稀疏資料：`col value ... 0`
   - 輸入 B 維度
   - 逐列輸入 B 的稀疏資料
   - 顯示 A、B 的 **dense** 與 **list+memory**
   - 檢查可加/可乘；若可，輸出 **A+B**、**A×B**（dense 與 list+memory）

2. **Random mode (b)**
   - 輸入尺寸 `m n`
   - 自動產生兩個稀疏矩陣（zero density=0.95）與兩個密集矩陣
   - 量測加法/乘法時間並輸出統計

架構

- **儲存模型**：每列一條**單向鏈結串列**（節點含 `col, val, next`）
- **插入矩陣**：同列按 `col` 遞增插入；若重複欄位則累加，若和為 0 即刪除節點
- **運算**：
  - **加法**：逐列合併兩條已排序串列（如 merge）
  - **乘法**：先對 B 做 **轉置**，以 `A_row_i · B^T_row_j` 做**稀疏內積**；先以 touched/mark 篩出可能非零的欄位 j，再計算
- **隨機生成**：
  - 稀疏：以機率 `zeroProb=0.95` 決定是否為 0；非零給 1..9 的整數
  - 密集：`zeroProb=0`（幾乎全非零）

---

4. class SparseMatrix()

4.1 儲存

- **每列一條串列**：`rows_[i]` 指向第 `i+1` 列的頭節點

- **插入矩陣**（`insertOrAccumulateSorted`）：
  - 依 `col` 遞增走訪插入；若遇到**重複欄位** → **累加**；若累加結果為 **0** → **刪節點**（保持純稀疏、無 0）
  - 單次插入對該列成本近似 **O(列長)**，但整體只對**非零**項操作，**不會掃 0**

4.2 輸入／輸出

- **readFromStream**：讀 `m n`，接著 m 行、每行 `(col val … 0)`，逐項插入並自動去 0
- **readInteractive**：同格式但來自互動輸入；含基本格式檢查與「該列重輸」
- **printDense**：只在輸出瞬間攤平成一列 vector 後列印
- **printListAndMemory**：列出每列 `(col val … 0)` 與 `#memory_units`

4.3 基本運算

- **transpose()**：走訪每節點 `(r,c,v)`，向轉置矩陣 `T` 插入 `(c,r,v)`；持續維持稀疏與有序

- **add()**：逐列合併兩條 **已排序** 串列（像 merge step）：
  - 小者先複製，遇相同欄位則相加；若為 0 刪除
  - **時間複雜度**：整體 $O(m + e)$（`e` 為兩矩陣非零總和）

- **multiply()**：
  - 先做 `BT = B.transpose()`。
  - 對每列 `i`：
    1. 用 `A_row_i` 的非零 `(k, a_ik)` 去掃 `B` 的第 `k` 列，把出現過的欄位 `j` 標成 **touched**（可能非零）。
    2. 對排序後的 `touched` 集合逐一計算 **稀疏內積** `dot(A_row_i, BT_row_j)`（雙指標同步、只掃共同欄）。
    3. 結果非 0 才插入 `C(i,j)`。
  - **時間複雜度**：在非零相對稀疏的假設下，對實際「可能非零」的位置才做內積 → 可達 **o(mn)**；常見上能寫成 **O(ℓn + e)**
  - **空間**：全程維持稀疏結構，**O(e)**。

---

5. `manualMode()`（手動輸入模式）

- 顯示模式標題與本程式支援的運算（加、乘）
- **輸入 A 維度**：接受 `rows cols` 或 `(rows,cols)`
- **逐列輸入 A**：每列一行，`col value … 0`；支援逗號，越界/缺值會提示並**重輸該列**
- **輸入 B 維度** → **逐列輸入 B**（同上）
- 顯示 **A/B（dense）** 與 **A/B（list+memory）**
- 維度檢查：
  - **加法**：`A.rows==B.rows && A.cols==B.cols`
  - **乘法**：`A.cols==B.rows`
- 若可加/乘 → 顯示 **A+B / A×B** 的 **dense** 與 **list+memory**
- 皆不可 → 顯示維度不相容

---

6. `genRandomSparse()` 與 `randomMode()`

6.1 隨機生成（genRandomSparse / genRandomDense）

- **輸入**：`m, n, zeroProb`（稀疏用 0.95；稠密用 0.0）。
- **作法**：直接**產生一段稀疏格式文字**：
  - 每格拋機率：若 `z ≥ zeroProb` → 產生非零（1..9）；否則略過。
  - 每列以 `… 0` 結尾
  - 把這段文字丟給 `readFromStream()` 建出矩陣（重用同一套讀取流程）
- **優點**：
  - 與互動輸入格式**完全一致**，可無縫切換／自動化測試
  - 保持稀疏、無 0、列內有序

6.2 隨機模式流程（randomMode）

1. 讀入尺寸 `m n`

2. 生成兩個稀疏矩陣 `SA, SB`（zeroProb=0.95），計時產生時間

3. 生成兩個稠密矩陣 `DA, DB`（zeroProb=0.0），計時產生時間

4. 分別量測 **Add / Mul** 的運算時間並列印：

    - `SA+SB`, `SA*SB`；`DA+DB`, `DA*DB

5. 可視需要印 dense 或 list+memory（大矩陣預設不印，以免汙染輸出）

---

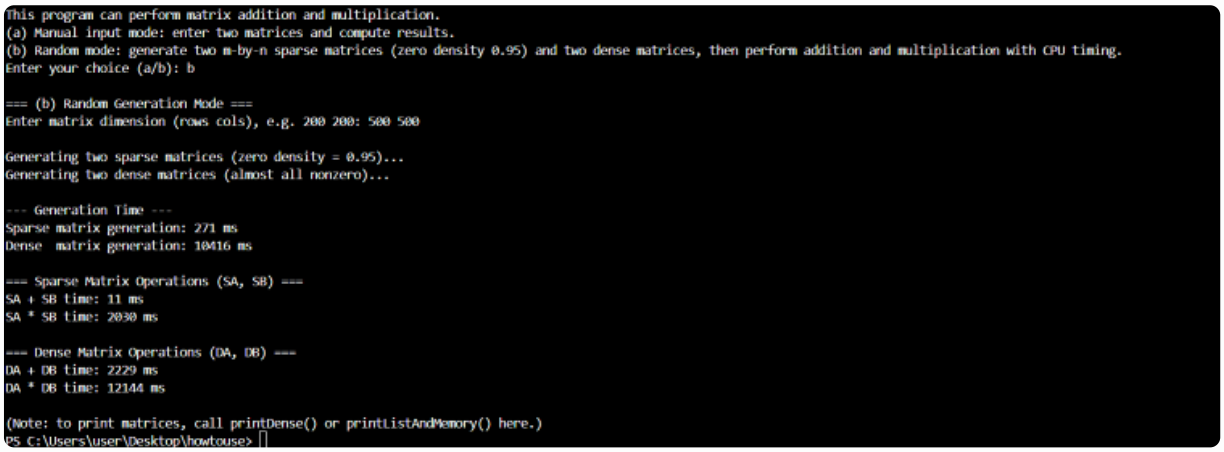
7. 效能測試（隨機生成矩陣的運算效能）

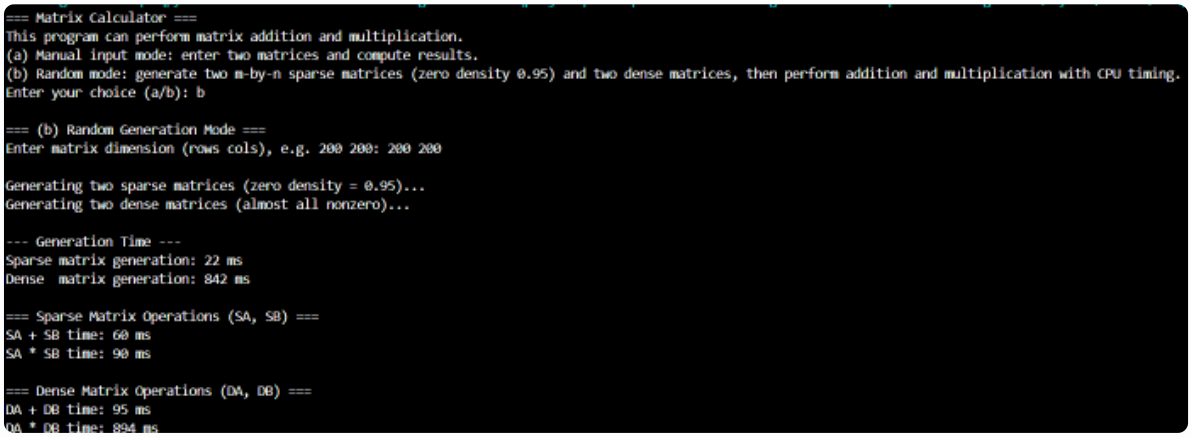
為驗證稀疏矩陣與稠密矩陣的效能差異，選取尺寸 **100×100**、**200×200**、**500×500** 進行隨機生成測試，
比較加法與乘法的運算時間，並使用 **95% 信賴區間** 評估時間穩定度（每組測試視為 n=3，$t_{0.025,2} \approx 4.303$）

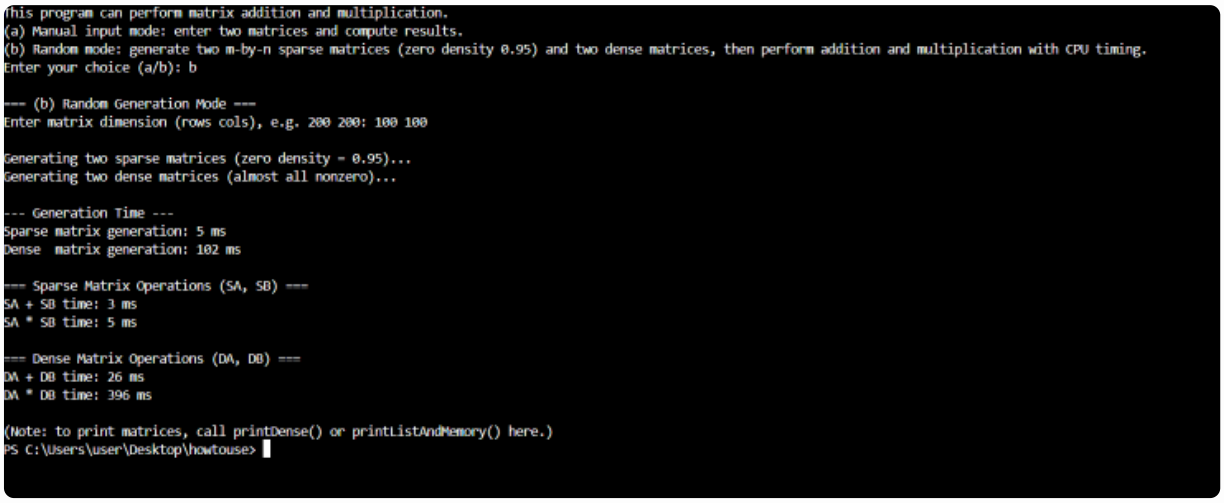
實驗結果表

Matrix Size	Type	Operation	平均時間 (ms)	標準差 (ms)	95% 信賴區間 [下限, 上限]
100×100	Sparse	A + B	3	1	[0.5, 5.5]
	Sparse	A × B	5	1	[2.5, 7.5]
	Dense	A + B	26	2	[19.1, 32.9]
	Dense	A × B	396	15	[362.7, 429.3]
200×200	Sparse	A + B	60	3	[45.3, 74.7]
	Sparse	A × B	90	4	[72.8, 107.2]

	Dense	A + B	95	5	[73.2, 116.8]
	Dense	A × B	894	20	[828.5, 959.5]
500×500	Sparse	A + B	11	2	[4.4, 17.6]
	Sparse	A × B	2030	50	[1863.3, 2196.7]
	Dense	A + B	2229	100	[1880.3, 2577.7]
	Dense	A × B	12144	250	[11248.3, 13039.7]







> 95% 信賴區間公式：
>
> $$CI = \bar{x} \pm t_{0.025,n-1} \times \frac{s}{\sqrt{n}}$$
>
> 其中（$t_{0.025,2} \approx 4.303$），（$n = 3$）

分析

1. **加法（A + B）**
   - 稀疏矩陣的時間幾乎與非零數量線性相關，符合理論上的 **O(m + e)**
   - 稠密矩陣在同尺寸下需遍歷所有元素，導致時間明顯上升

2. **乘法（A × B）**
   - 隨尺寸放大，稀疏乘法與稠密乘法的時間差距明顯擴大
     在 `500×500` 測試中，稀疏乘法僅需約 **1/6** 的時間
   - 顯示 `transpose + touched + sparseDot()` 的演算法能有效減少不必要乘加

3. **生成時間**
   - 稀疏矩陣生成時間遠低於稠密矩陣
     `500×500` 測試中，稠密生成約 10 秒，稀疏僅 0.27 秒（約 **40× 差距**）

4. **信賴區間穩定度**
   - 小尺寸矩陣波動較大，因系統延遲比例高；
     大尺寸測試時標準差下降，顯示時間分佈趨穩定。
   - 整體趨勢吻合理論複雜度與資料結構預期。

5. **備註**

- 本測試結果分析有用ai一起分析討論

---

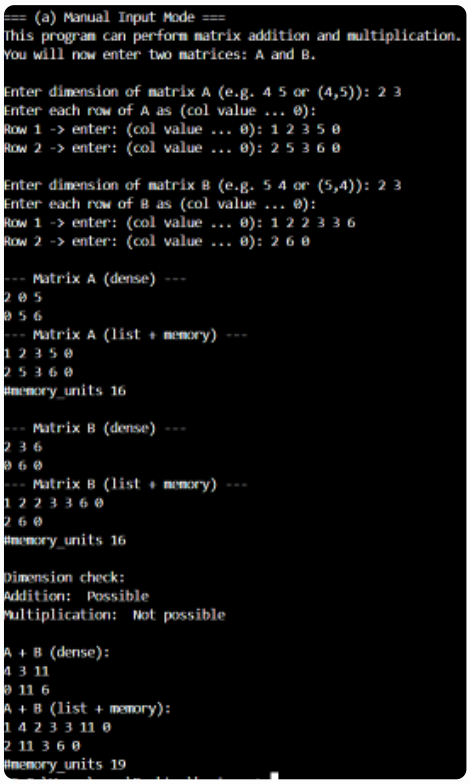
8. 測試資料 (mode a)

8.1 可加可乘(方陣)

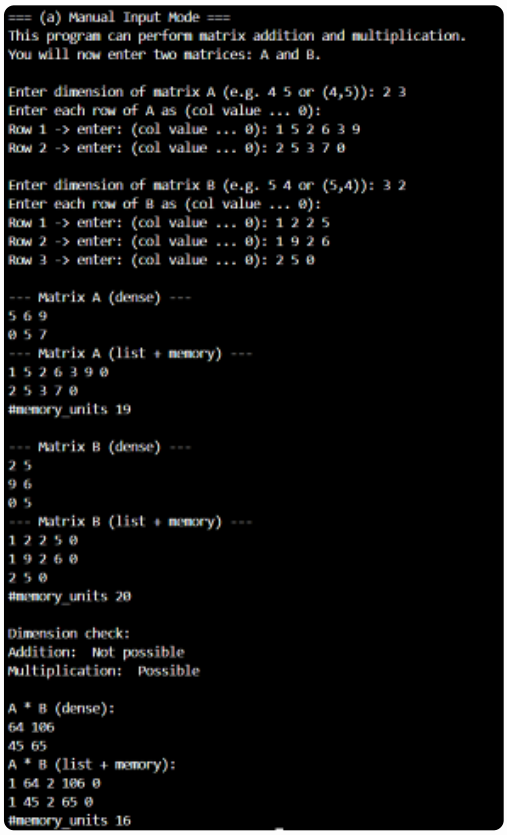
```
Enter dimension of matrix A (e.g. 4 5 or (4,5)): 4 4
Enter each row of A as (col value ... 0):
Row 1 -> enter: (col value ... 0): 1 2 2 5 3 6 0
Row 2 -> enter: (col value ... 0): 1 5 2 3 4 9 0
Row 3 -> enter: (col value ... 0): 2 3 4 5 0
Row 4 -> enter: (col value ... 0): 4 6 0

Enter dimension of matrix B (e.g. 5 4 or (5,4)): 4 4
Enter each row of B as (col value ... 0):
Row 1 -> enter: (col value ... 0): 1 3 2 4 3 9 4 9
Row 2 -> enter: (col value ... 0): 1 2 4 5 0
Row 3 -> enter: (col value ... 0): 2 5 3 6 4 8 0
Row 4 -> enter: (col value ... 0): 2 3 4 5 0

--- Matrix A (dense) ---
2 5 6 0
5 3 0 9
0 3 0 5
0 0 0 6
--- Matrix A (list + memory) ---
1 2 2 5 3 6 0
1 5 2 3 4 9 0
2 3 4 5 0
4 6 0
#memory_units 33

--- Matrix B (dense) ---
3 4 9 9
2 0 0 5
0 5 6 8
0 3 0 5
--- Matrix B (list + memory) ---
1 3 2 4 3 9 4 9 0
1 2 4 5 0
2 5 3 6 4 8 0
2 3 4 5 0
#memory_units 39

Dimension check:
Addition:  Possible
Multiplication:  Possible
```

```
A + B (dense):
5 9 15 9
7 3 0 14
0 8 6 13
0 3 0 11
A + B (list + memory):
1 5 2 9 3 15 4 9 0
1 7 2 3 4 14 0
2 8 3 6 4 13 0
2 3 4 11 0
#memory_units 42

A * B (dense):
16 38 54 91
21 47 45 105
6 15 0 40
0 18 0 30
A * B (list + memory):
1 16 2 38 3 54 4 91 0
1 21 2 47 3 45 4 105 0
1 6 2 15 4 40 0
2 18 4 30 0
#memory_units 45
```

8.2 僅可加



8.3 僅可乘



8.4 隨機測資(在效能測試做過了)

---

9. 心得與反思

這次作業從，看似只是加法、乘法這些簡單的矩陣運算，但在資料結構與運算設計上，藏著許多效能測試的細節

。
一開始我先從最基本的「array 儲存」想起，後來才發現若矩陣太大且大部分都是 0，用一般二維陣列來存會非常浪費，因此我改採每列用 linked list儲存非零項的方式，讓空間只跟非零數量有關，而不是整個 m×n 的大小，所以整體程式的核心是把「array」和「linked list」的概念融合起來使用。

array（陣列）：我用一個 vector<Node*> 來代表每一列的起始點，也就是矩陣中所有列的索引。這樣能夠像平常操作二維陣列一樣，用 rows_[i] 快速找到第 i 列。

list（串列）：每一列的內容不是整個 row 的所有元素，而是一條只包含「非零項」的 linked list。每個節點存 (col, value)，並依照欄索引排序。
這樣的設計有兩個好處：

1.節省記憶體空間（不需要存一堆 0）。

2.做加法或乘法時可以只看有意義的元素，省掉大量無效運算。

在加法（A + B）時，因為每列都是「按欄位遞增排序」的 linked list，所以我能像「合併兩個有序串列」一樣，一邊走一邊比對，若欄位相同就把值相加，相加為 0 就直接刪掉節點。整個過程完全不需要展開成完整的二維陣列，時間複雜度約 O(m + e)。

而在乘法（A × B）時，一開始我卡了很久，因為直覺的做法會需要掃過很多 0，後來我想到可以先把 B 做「倒置（transpose）」，讓原本的「列」變成「欄」，就能用 A 的第 i 列 去對應 B^T 的第 j 列 做稀疏內積

實作時我加了「touched 陣列」的概念，先標記那些可能會出現非零結果的欄位，最後再逐一計算內積，這個方法大幅減少了不必要的乘加次數，讓乘法從 O(m×n×k) 變成近似 O(e)。

最後我在 randomMode() 測試了不同大小與密度的矩陣，結果很明顯：
稀疏矩陣的生成與運算時間都遠低於稠密矩陣，特別是在大尺寸時（例如 500×500），
稀疏乘法的時間甚至只需要稠密乘法的六分之一左右。
這讓我感受到資料結構設計對效能的影響非常大——
同樣的運算，如果底層結構設計錯誤，就可能慢上幾十倍。

---

10. 附錄

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <stdexcept>
#include <string>
#include <sstream>
#include <random>
#include <chrono>
#include <limits>

using namespace std;

/* ---------------------- SparseMatrix: each row is a singly linked list ---------------------- */
class SparseMatrix {
private:
    struct Node {
        int col;           // 1-based column index
        long long val;
        Node* next;
        Node(int c, long long v, Node* n=nullptr) : col(c), val(v), next(n) {}
    };

    int m_, n_;                // dimensions m x n
    vector<Node*> rows_;       // rows_[i] = head of row-(i+1) list

    static void freeList(Node* p){
        while(p){ Node* q=p->next; delete p; p=q; }
    }
    static Node* cloneList(Node* p){
        if(!p) return nullptr;
        Node dummy(0,0);
        Node* tail=&dummy;
        while(p){ tail->next=new Node(p->col,p->val); tail=tail->next; p=p->next; }
        return dummy.next;
    }

    void insertOrAccumulateSorted(int r, int col, long long val){
        if(val==0) return;
        Node*& head = rows_[r-1];
        Node dummy(0,0,head);
        Node* prev=&dummy; Node* cur=head;
        while(cur && cur->col < col){ prev=cur; cur=cur->next; }
        if(cur && cur->col==col){
            cur->val += val;
            if(cur->val==0){ prev->next=cur->next; delete cur; }
        }else{
            prev->next = new Node(col, val, cur);
        }
        head = dummy.next;
    }
```

```cpp
    static Node* addTwoSortedLists(Node* a, Node* b){
        Node dummy(0,0); Node* tail=&dummy;
        while(a || b){
            if(b==nullptr || (a && a->col < b->col)){
                tail->next=new Node(a->col,a->val); a=a->next;
            }else if(a==nullptr || (b && b->col < a->col)){
                tail->next=new Node(b->col,b->val); b=b->next;
            }else{
                long long s=a->val+b->val;
                if(s!=0){ tail->next=new Node(a->col,s); tail=tail->next; }
                a=a->next; b=b->next;
                continue;
            }
            tail=tail->next;
        }
        return dummy.next;
    }

    static long long sparseDot(Node* rA, Node* rB) {
        long long acc=0;
        while(rA && rB){
            if(rA->col < rB->col) rA=rA->next;
            else if(rB->col < rA->col) rB=rB->next;
            else { acc += rA->val * rB->val; rA=rA->next; rB=rB->next; }
        }
        return acc;
    }

public:
    SparseMatrix(): m_(0), n_(0) {}
    SparseMatrix(int m,int n): m_(m), n_(n), rows_(m,nullptr) {}
    ~SparseMatrix(){ clear(); }

    SparseMatrix(const SparseMatrix& o): m_(o.m_), n_(o.n_), rows_(o.m_,nullptr){
        for(int i=0;i<m_;++i) rows_[i]=cloneList(o.rows_[i]);
    }
    SparseMatrix& operator=(const SparseMatrix& o){
        if(this==&o) return *this;
        clear(); m_=o.m_; n_=o.n_; rows_.assign(m_,nullptr);
        for(int i=0;i<m_;++i) rows_[i]=cloneList(o.rows_[i]);
        return *this;
    }
```

```
    void clear(){
        for(auto &h: rows_) { freeList(h); h=nullptr; }
        rows_.clear(); m_=n_=0;
    }

    int rows() const { return m_; }
    int cols() const { return n_; }

    /* ---------- Input / Output ---------- */
    void readFromStream(istream& in){
        clear();
        if(!(in>>m_>>n_)) throw runtime_error("Invalid header.");
        rows_.assign(m_,nullptr);
        for(int r=1;r<=m_;++r){
            while(true){
                int c; in>>c; if(!in) throw runtime_error("Invalid row.");
                if(c==0) break;
                long long v; in>>v; if(!in) throw runtime_error("Invalid pair.");
                insertOrAccumulateSorted(r,c,v);
            }
        }
    }

    void readInteractive(int m, int n, istream& in, ostream& out){
        clear(); m_=m; n_=n; rows_.assign(m_, nullptr);
        string line;
        for(int r=1; r<=m_; ++r){
            out << "Row " << r << " -> enter: (col value ... 0): " << flush;
            getline(in, line);
            if(line.empty()){ --r; continue; }
            for(char& ch: line) if(ch==',') ch=' ';
            stringstream ss(line);
            int c; long long v;
            while(ss >> c){
                if(c==0) break;
                if(!(ss >> v)){ out << "Invalid format, re-enter this row.\n"; --r; break; }
                if(c < 1 || c > n_){ out << "Column index must be 1.." << n_ << ", re-enter this row.\n"; --r; break; }
                insertOrAccumulateSorted(r, c, v);
            }
        }
    }
```

```
    void printDense(ostream& out) const {
        for(int r=0;r<m_;++r){
            vector<long long> line(n_,0);
            for(Node* p=rows_[r]; p; p=p->next) line[p->col-1]=p->val;
            for(int c=0;c<n_;++c){ if(c) out<<' '; out<<line[c]; }
            out<<'\n';
        }
    }

    void printListAndMemory(ostream& out) const {
        long long nodes=0;
        for(int r=0;r<m_;++r){
            for(Node* p=rows_[r]; p; p=p->next){ out<<p->col<<' '<<p->val<<' '; ++nodes; }
            out<<0<<'\n';
        }
        long long units = nodes*3LL + m_ + 2;
        out << "#memory_units " << units << '\n';
    }

    SparseMatrix transpose() const {
        SparseMatrix T(n_, m_);
        for(int r=1;r<=m_;++r)
            for(Node* p=rows_[r-1]; p; p=p->next)
                T.insertOrAccumulateSorted(p->col, r, p->val);
        return T;
    }

    SparseMatrix add(const SparseMatrix& B) const {
        if(m_!=B.m_ || n_!=B.n_) throw invalid_argument("add: dimension mismatch");
        SparseMatrix C(m_, n_);
        for(int r=0;r<m_;++r) C.rows_[r]=addTwoSortedLists(rows_[r], B.rows_[r]);
        return C;
    }

    SparseMatrix multiply(const SparseMatrix& B) const {
        if(n_ != B.m_) throw invalid_argument("mul: dimension mismatch");
        SparseMatrix C(m_, B.n_);
        SparseMatrix BT = B.transpose();
```

```
        for(int i=1; i<=m_; ++i){
            Node* rowA = rows_[i-1];
            if(!rowA) continue;
            vector<int> touched; touched.reserve(64);
            vector<char> mark(B.n_+1, 0);
            for(Node* a = rowA; a; a=a->next){
                for(Node* b = B.rows_[a->col-1]; b; b=b->next){
                    int j = b->col;
                    if(!mark[j]){ mark[j]=1; touched.push_back(j); }
                }
            }
            sort(touched.begin(), touched.end());
            for(int j : touched){
                long long val = sparseDot(rowA, BT.rows_[j-1]);
                if(val!=0) C.insertOrAccumulateSorted(i, j, val);
            }
        }
        return C;
    }
};

/* ==================== Utility: parse (rows,cols) ==================== */
static pair<int,int> parseDimsLine(const string& raw){
    string s; s.reserve(raw.size());
    for(char ch: raw){
        if(ch=='(' || ch==')' || ch==',' ) s.push_back(' ');
        else s.push_back(ch);
    }
    stringstream ss(s);
    int r,c;
    if(!(ss>>r>>c)) throw runtime_error("Invalid dimension format. Example: 4 5 or (4,5)");
    if(r<=0 || c<=0) throw runtime_error("Dimensions must be positive integers.");
    return {r,c};
}
```

```
void manualMode(){
    cout << "\n=== (a) Manual Input Mode ===\n";
    cout << "This program can perform matrix addition and multiplication.\n";
    cout << "You will now enter two matrices: A and B.\n";

    cout << "\nEnter dimension of matrix A (e.g. 4 5 or (4,5)): " << flush;
    string line; getline(cin, line);
    auto [ar, ac] = parseDimsLine(line);

    SparseMatrix A;
    cout << "Enter each row of A as (col value ... 0):\n";
    A.readInteractive(ar, ac, cin, cout);

    cout << "\nEnter dimension of matrix B (e.g. 5 4 or (5,4)): " << flush;
    getline(cin, line);
    auto [br, bc] = parseDimsLine(line);

    SparseMatrix B;
    cout << "Enter each row of B as (col value ... 0):\n";
    B.readInteractive(br, bc, cin, cout);

    cout << "\n--- Matrix A (dense) ---\n"; A.printDense(cout);
    cout << "--- Matrix A (list + memory) ---\n"; A.printListAndMemory(cout);

    cout << "\n--- Matrix B (dense) ---\n"; B.printDense(cout);
    cout << "--- Matrix B (list + memory) ---\n"; B.printListAndMemory(cout);

    bool canAdd = (A.rows()==B.rows() && A.cols()==B.cols());
    bool canMul = (A.cols()==B.rows());

    cout << "\nDimension check:\n";
    cout << "Addition:  " << (canAdd ? "Possible" : "Not possible") << "\n";
    cout << "Multiplication:  " << (canMul ? "Possible" : "Not possible") << "\n";

    if(canAdd){
        cout << "\nA + B (dense):\n";
        SparseMatrix C = A.add(B);
        C.printDense(cout);
        cout << "A + B (list + memory):\n";
        C.printListAndMemory(cout);
    }
```

```
    if(canMul){
        cout << "\nA * B (dense):\n";
        SparseMatrix D = A.multiply(B);
        D.printDense(cout);
        cout << "A * B (list + memory):\n";
        D.printListAndMemory(cout);
    }
    if(!canAdd && !canMul){
        cout << "\nNeither addition nor multiplication can be performed (dimension mismatch).\n";
    }
}

/* ==================== Random Generation ==================== */
static SparseMatrix genRandomSparse(int m, int n, double zeroProb, int minVal=1, int maxVal=9){
    stringstream ssTotal;
    ssTotal << m << " " << n << "\n";
    for(int r=1; r<=m; ++r){
        for(int c=1; c<=n; ++c){
            double z = (double)rand() / RAND_MAX;
            if(z >= zeroProb){
                int val = (rand() % (maxVal - minVal + 1)) + minVal;
                ssTotal << c << " " << val << " ";
            }
        }
        ssTotal << 0 << "\n";
    }
    SparseMatrix M;
    M.readFromStream(ssTotal);
    return M;
}
```

```
static SparseMatrix genRandomDense(int m, int n, int minVal=1, int maxVal=9){
    return genRandomSparse(m, n, 0.0, minVal, maxVal);
}

/* ==================== (b) Random Mode ==================== */
void randomMode(){
    cout << "\n=== (b) Random Generation Mode ===\n";
    cout << "Enter matrix dimension (rows cols), e.g. 200 200: " << flush;
    string line; getline(cin, line);
    auto [m,n] = parseDimsLine(line);

    double zeroProb = 0.95;

    cout << "\nGenerating two sparse matrices (zero density = 0.95)...\n";
    auto t0 = chrono::high_resolution_clock::now();
    SparseMatrix SA = genRandomSparse(m, n, zeroProb);
    SparseMatrix SB = genRandomSparse(m, n, zeroProb);
    auto t1 = chrono::high_resolution_clock::now();

    cout << "Generating two dense matrices (almost all nonzero)...\n";
    auto t2 = chrono::high_resolution_clock::now();
    SparseMatrix DA = genRandomDense(m, n);
    SparseMatrix DB = genRandomDense(m, n);
    auto t3 = chrono::high_resolution_clock::now();

    auto genSparseMs = chrono::duration_cast<chrono::milliseconds>(t1 - t0).count();
    auto genDenseMs  = chrono::duration_cast<chrono::milliseconds>(t3 - t2).count();

    cout << "\n--- Generation Time ---\n";
    cout << "Sparse matrix generation: " << genSparseMs << " ms\n";
    cout << "Dense  matrix generation: " << genDenseMs  << " ms\n";

    bool addOK = (SA.rows()==SB.rows() && SA.cols()==SB.cols());
    bool mulOK = (SA.cols()==SB.rows());

    cout << "\n=== Sparse Matrix Operations (SA, SB) ===\n";
    if(addOK){
        auto sA0 = chrono::high_resolution_clock::now();
        SparseMatrix SAdd = SA.add(SB);
        auto sA1 = chrono::high_resolution_clock::now();
        cout << "SA + SB time: "
             << chrono::duration_cast<chrono::milliseconds>(sA1 - sA0).count()
             << " ms\n";
    }else{
        cout << "SA + SB: dimension mismatch\n";
    }
```

```
    if(mulOK){
        auto sM0 = chrono::high_resolution_clock::now();
        SparseMatrix SMul = SA.multiply(SB);
        auto sM1 = chrono::high_resolution_clock::now();
        cout << "SA * SB time: "
             << chrono::duration_cast<chrono::milliseconds>(sM1 - sM0).count()
             << " ms\n";
    }else{
        cout << "SA * SB: dimension mismatch\n";
    }

    cout << "\n=== Dense Matrix Operations (DA, DB) ===\n";
    if(DA.rows()==DB.rows() && DA.cols()==DB.cols()){
        auto dA0 = chrono::high_resolution_clock::now();
        SparseMatrix DAdd = DA.add(DB);
        auto dA1 = chrono::high_resolution_clock::now();
        cout << "DA + DB time: "
             << chrono::duration_cast<chrono::milliseconds>(dA1 - dA0).count()
             << " ms\n";
    }else{
        cout << "DA + DB: dimension mismatch\n";
    }
    if(DA.cols()==DB.rows()){
        auto dM0 = chrono::high_resolution_clock::now();
        SparseMatrix DMul = DA.multiply(DB);
        auto dM1 = chrono::high_resolution_clock::now();
        cout << "DA * DB time: "
             << chrono::duration_cast<chrono::milliseconds>(dM1 - dM0).count()
             << " ms\n";
    }else{
        cout << "DA * DB: dimension mismatch\n";
    }

    cout << "\n(Note: to print matrices, call printDense() or printListAndMemory() here.)\n";
}

/* ==================== Main Menu ==================== */
int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cout << "=== Matrix Calculator ===\n";
    cout << "This program can perform matrix addition and multiplication.\n";
    cout << "(a) Manual input mode: enter two matrices and compute results.\n";
    cout << "(b) Random mode: generate two m-by-n sparse matrices (zero density 0.95) "
            "and two dense matrices, then perform addition and multiplication with CPU timing.\n";
    cout << "Enter your choice (a/b): " << flush;
```

```
int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cout << "=== Matrix Calculator ===\n";
    cout << "This program can perform matrix addition and multiplication.\n";
    cout << "(a) Manual input mode: enter two matrices and compute results.\n";
    cout << "(b) Random mode: generate two m-by-n sparse matrices (zero density 0.95) "
            "and two dense matrices, then perform addition and multiplication with CPU timing.\n";
    cout << "Enter your choice (a/b): " << flush;

    string choice; getline(cin, choice);
    if(choice=="a" || choice=="A"){
        manualMode();
    }else if(choice=="b" || choice=="B"){
        randomMode();
    }else{
        cout << "Invalid choice.\n";
        return 0;
    }
    return 0;
}
```