CS 124 Programming Assignment 2: Spring 2023

Your name(s) (up to two): Nicholas Lyu, Jacqueline Liu
No. of late days used on previous psets: 3
No. of late days used after including this pset: 0

Task 1: Cross-over point estimate

Consider the following implementation of simple matrix multiplication (implementation by definition):

```
template <class T>
Matrix<T> Matrix<T>::simplematmul(const Matrix& Y){
    assert (ncols == nrows);
    Matrix<T> C(nrows, ncols, (T)(0));
    for (unsigned int i=0; i<nrows; i++)
        for (unsigned int k=0; k<ncols; k++)
            for (unsigned int j=0; j<Y.ncols; j++)
                C(i, j) += M[i][k] * Y.M[k][j];
    return C;
}
```

Given $D \times D$ square matrices X, Y (X is implicit in invocation of X.matmul(Y)), there are D^3 scalar multiplications and $D^3 - D^2$ additions (assuming, as per question, that all other operations are free). Thus total runtime is $T_1(D) = 2D^3 - D^2$.

Consider the following implementation of strassen matmul:

```
Matrix<T> Matrix<T>::strassen(const Matrix<T>& Y){
    assert (ncols == Y.nrows);
    if (nrows == 1 && Y.ncols == 1)
        return dot(*this, Y);
    // We only implement for square matrices
    if (ncols % 2 != 0 || nrows % 2 != 0 || Y.ncols % 2 != 0){
        return pad().strassen(Y.pad(), threshold).slice(0, -1, 0, -1);
    }
    Matrix<T> A = slice(0, nrows / 2, 0, ncols / 2), B = slice(0, nrows / 2, ncols / 2, ncols),
              C = slice(nrows / 2, nrows, 0, ncols / 2),
              D = slice(nrows / 2, nrows, ncols / 2, ncols),
              E = Y.slice(0, Y.nrows / 2, 0, Y.ncols / 2),
              F = Y.slice(0, Y.nrows / 2, Y.ncols / 2, Y.ncols),
              G = Y.slice(Y.nrows / 2, Y.nrows, 0, Y.ncols / 2),
              H = Y.slice(Y.nrows / 2, Y.nrows, Y.ncols / 2, Y.ncols);
    auto P1 = A.strassen(F-H), P2 = (A+B).strassen(H),
         P3 = (C+D).strassen(E), P4 = D.strassen(G-E),
         P5 = (A+D).strassen(E+H);
    // Use in-place operations as much as possible to avoid memory allocation
    B -= D;
```

```
    G += H;
    auto P6 = B.strassen(G);
    C -= A;
    E += F;
    auto P7 = C.strassen(E);
    P6 += P4; // P6 := P6 + P5 + P4 - P2
    P6 += P5;
    P6 -= P2;
    P2 += P1; // P2 := P2 + P1
    P4 += P3; // P4 := P3 + P4
    P1 -= P3; // P1 := P1 - P3 + P65 + P7
    P1 += P5;
    P1 += P7;
    return ((P6).append(P2, 1)).append((P4).append(P1, 1), 0);
}
```

We first differentiate between in-place and out-of-place element-wise matrix operations (for more detailed discussion consult *strassen implementation* section in Task 2 discussion below):

- **In-place operations**: in-place operations are of form $A+ = B$, they have n^2 elementary operations.
- **Out-of-place operations**: out-of-place operations are of form $C = A + B$, they have $2n^2$ cost. In particular, chains of out-of-place operations such as $D = A + B + C = (A + B) + C$ equivalent to $T = A + B$; $D + T + C$ have $4n^2$ cost. Each out-of-place operation invokes $2n^2$ cost. Padding and slicing are both out-of-place operations since they return distinct objects. The implementations of slicing and appending (padding) uses n^2 elementary operations:

```
    Matrix<T> Matrix<T>::slice(int ri, int rj, int ci, int cj) const{
        rj = rj < 0 ? nrows + rj : rj;
        cj = cj < 0 ? ncols + cj : cj;
        ri = ri < 0 ? nrows + ri : ri;
        ci = ci < 0 ? ncols + ci : ci;
        assert (ri >= 0 && ri < rj && (unsigned int) rj <= nrows && ci >= 0 && ci < cj
                && (unsigned int) cj <= ncols);
        Matrix<T> B(rj-ri, cj-ci, (T) 0);
        for (unsigned int i=0; i<B.nrows; ++i)
            for (unsigned int j=0; j<B.ncols; ++j)
                B(i, j) += M[i+ri][j+ci];
        return B;
    }

    Matrix<T> Matrix<T>::append(const Matrix<T>& B, unsigned int axis) const {
        assert (axis == 0 || axis == 1);
        unsigned int nr = nrows + (1-axis) * B.nrows, nc = ncols + axis * B.ncols;
        Matrix<T> C(nr, nc);

        for (unsigned int i=0; i<nr; i++)
            for (unsigned int j=0; j<nc; j++){
```

```
22                  if (i >= nrows){
23                      C(i, j) = B.M[i-nrows][j];
24                      continue;
25                  }
26                  if (j >= ncols){
27                      C(i, j) = B.M[i][j-ncols];
28                      continue;
29                  }
30                  C(i, j) = M[i][j];
31              }
32          return C;
33      }
34
35      Matrix<T> Matrix<T>::pad() const{
36          return append(*(new Matrix<T>(1, ncols, (T)0)), 0).append(
37                  *(new Matrix<T>(nrows+1, 1, (T)0)), 1);
38      }
```

We now do a line-by-line runtime analysis of Strassen above: assuming X, Y have even size $n = 2d$

- Slicing in L9-15 invokes $8d^2$ cost.
- L16-18 includes 6 out-of-place operations and 5 matrix multiplications, invoking $12d^2 + 5T_2(d)$ cost.
- L20-33 includes 12 in-place operations and 2 matrix multiplications, invoking $12d^2 + 2T_2(d)$ cost.
- L34 invokes two appends resulting in $d \times 2d$ matrices and one append resulting in $2d \times 2d$ matrix, invoking $2 \cdot 2d^2 + (2d)^2 = 8d^2$ cost.

The above analysis yields $T_2(2d) = 7T_2(d) + 40d^2$. Let us turn our attention now to odd case: L7 given input matrix of size $2d - 1$ invokes two pads on matrix of size $2d$ and a slice on matrix of size $2d$. Total cost is then $T_2(2d - 1) = T_2(2d) + 3(2d)^2 = 7T_2(d) + 52d^2$. In general, the time-recurrence relation for strassen is of form $T_2(2d) = 7T_2(d) + c_1 d^2$ and $T_2(2d - 1) = 7T_2(d) + c_2 d^2$ for constants c_1, c_2 depending on engineering details.

The fundamental relation for computing crossover point is as follows: $\begin{cases} T_1(2n) \geq 7T_1(n) + c_1 n^2 \\ T_2(2n-1) \geq 7T_1(n) + c_2 n^2 \end{cases}$
LHS is cost of computing larger matrix using simple matmul, and RHS is cost of computing matrix of larger step using a strassen split step (of overhead $c_j n^2$) and using simple matmul on the invoked smaller matrix multiplications. The even and odd crossoverpoints are $n^*_{\text{odd}} = 2n^0_{\text{odd}} - 1 \quad n^*_{\text{even}} = 2n^0_{\text{even}}$ where n^0 are the even / odd solutions to the equations above.

A simple analysis of Strassen which does not account for additional costs invoked in out-of-place operations and costs of padding and slicing yields $c_1 = c_2 = 18$. This yields crossover points $15, 38$ for even and odd-sized matrices, respectively. In particular this means:

- If matrix has even size n, call simple matmul if $n \leq 15$.
- If matrix has odd size n, call simple matmul if $n \leq 38$.

We also did a more detailed analysis of our strassen implementation with $c_1 = 40, c_2 = 52$ yielding even and odd crossover points $61, 73$, respectively. Our experimental results found the optimal crossover point to be much closer to 64, corroborating the relative accuracy of a more comprehensive analysis. However, we note that the simple analysis suffices to provide theoretical values for ideal implementations.

Task 2: Strassen Implementation, experimental cross-over point

Simple matmul implementation

Our code directly implements the definition $(AB)_{ij} = \sum_k A_{ik}B_{ij}$. Two optimizations are in order:

- Passing argument Y by constant reference to avoid copying large object (though matrix object is not so large anyways since we're storing array content on the heap)
- Swapping order of j, k loop to k, j loop in L6-7: this is more cache-friendly since $Y_{k,j}$ and $Y_{(k+1),j}$ are further apart in memory than $Y_{k,j}$ and $Y_{k,j+1}$

Strassen implementation

Our code implements strassen procedure as given in lecture notes. Several implementation details are in order:

- Matrix objects are passed by constant reference.
- Below is our implementation of padding: if the square matrix which called pad does not have even dimension size, pad with a trailing zero column and row to make it an even square.

```
1  Matrix<T> Matrix<T>::pad() const{
2      assert (nrows == ncols);
3      if (nrows % 2 == 0){
4          auto t = *this;
5          return t;
6      }
7      return append(*(new Matrix<T>(1, ncols, (T)0)), 0).append(
8              *(new Matrix<T>(nrows+1, 1, (T)0)), 1);
9  }
```

  Thus L9-12 in the Strassen implementation ensures that following procedure always operate on even dimension matrices. L11 trims off padded row and column to ensure algorithm correctness.

- We note subtle differences between in-place and out-of-place operations. In-place element-wise operations such as $A+ = B$ are implemented as follows:

```
1  Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& A)
2  {
3      assert (nrows == A.nrows && ncols == A.ncols);
4      for (unsigned int i = 0; i < nrows; ++i)
5          for (unsigned int j = 0; j < ncols; ++j)
```

```
            M[i][j] += A.M[i][j];
    return *this;
}
```

Note that this only invokes n^2 elementary operations. By contrast, out-of-place operations $A = A+B$ invokes additional object allocation since $(A + B)$ is a distinct object from A.

```
Matrix<T>::Matrix(const Matrix<T>& A){
    this->nrows = A.nrows;
    this->ncols = A.ncols;
    alloc();
    initialized = true;
    for (unsigned int i=0;i<nrows; i++)
        for (unsigned int j=0; j<ncols; j++)
            M[i][j] = A.M[i][j];
}

Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B){
    Matrix<T> C(A);
    return (C += B);
}
```

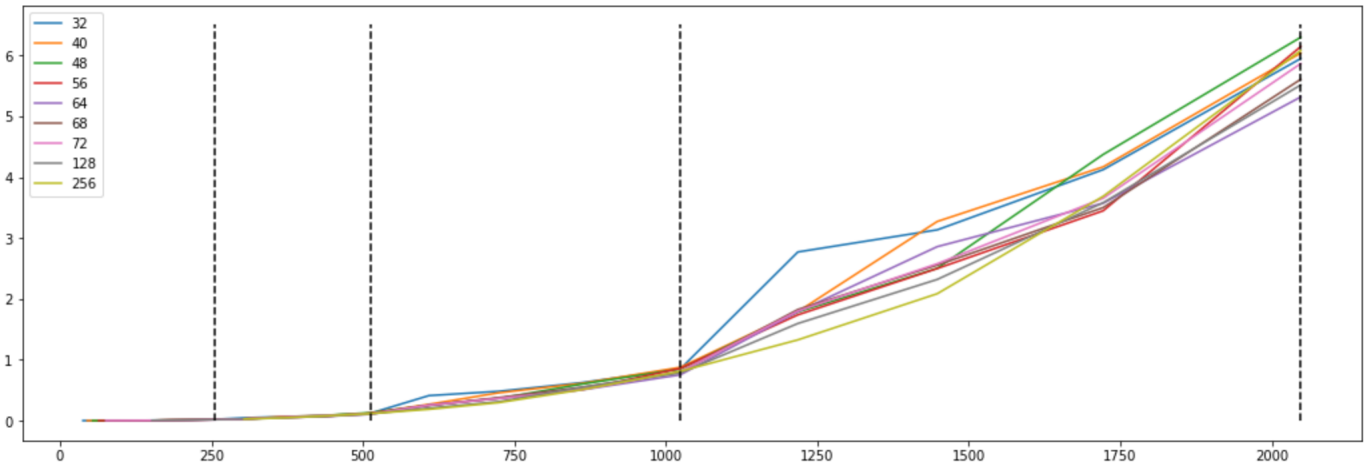
An out-of-place operation invokes $2n^2$ elementary operations. So our implementation of strassen avoids out-of-place operations as much as possible: L29-36 avoided 8 out-of-place operations, one per element-wise matrix operation in comment, similarly L23-27 avoid 4 out-of-place operations.

Experimental Cross-over point

The experimental cross-over point is not exactly a well-defined quantity: in particular, for cross-over points λ, λ', $T_\lambda(n) < T_{\lambda'}(n)$ does not necessarily imply $T_\lambda(n + \delta) < T_{\lambda'}(n + \delta)$ (though this relation is more or less guaranteed in asymptotic case $\delta \to \infty$. This is due to several reasons including, but not limited, to the ones below:

- Internal stochasticity of runtime: due to differences in computer loads, the same invocation may take different time. We attempt to mitigate this error by reporting runtime averaged over 10 runs.
- Caching behavior: due to system architecture, the caching behavior is opaque. The size of the matrix arguments may have large effect on runtime.
- Opaque stack invocation costs: our analysis did not account for costs associated with recursive calling, which are, again, dependent on systems architecture.

The figure below demonstrates runtime for different cross-over points on input matrix of varying sizes, varied over 10 runs:



The figure corroborates our claim that runtime do not display monotonic dominant behavior. However we can see that the optimal crossover point is around 64. This is relatively close to our theoretical value.

Task 3: Triangle in random graphs

We first implemented a random-fill function. When given an adjacency matrix A, it can randomly fill the matrix (non-diagonal) with probability p. Note that A is a symmetric matrix, and all the diagonal entries are zero, so we make sure A(i, i) = 0 and A(i, j) = A(j, i). The probability is implemented by generating a random number, and if its proportion to the RANDMAX is less than the given probability, then we determine it has an edge, otherwise no edge. Here we assume uniform distribution.

```
float prob[5] ={0.01,0.02,0.03,0.04,0.05};

void randomfill(Matrix<int>& A, float p){
    for(int i = 0; i < A.nrows; i++){
        A(i, i) = 0;
        for(int j = i+1; j < A.nrows; j++){
            float prob = (float) rand() / RAND_MAX;
            A(i, j) = (prob < p) ? 1 : 0;
            A(j, i) = A(i, j);
        }
    }
}
```

We also use the following fast-exponentiation algorithm for computing powers of a given matrix (though we note that this yields negligible advantage for small powers).

```
Matrix<T> Matrix<T>::pow(unsigned int k) {
    assert (nrows == ncols); // we can only exponentiate square matrix
    unsigned int n = nrows;

    // Starts from identity matrix
    auto A = Matrix<T>(n, n, (T) 0);
```

```
    for (unsigned int i=0; i<n; i++)
        A(i, i) = (T) 1;
    vector<Matrix<T> > v;
    v.push_back(*this);
    // Compute power-two exponents of current matrix
    while (powf(2.0, (float) v.size()) - 1 < (float) k)
        v.push_back(v[v.size()-1].strassen(v[v.size()-1]));

    // Accumulate relevant power-two exponents
    for (unsigned long i=0; i<v.size(); i++)
        if (((k & (1 << i)) >> i))
            A = A.strassen(v[i]);
    return A;
}

```

To generate real-random probability, we use srand function with the seed of current time. Then we use randomfill() function to generate a random matrix, use pow(3) function to raise A to A^3, and add up entries on the diagonal, which represent number of path from entry j to j. Then we divide the sum by 6 to get rid of duplicates, and get the number of triangles in the graph.

```
int main(){
    for(int i = 0; i < 5; i++){
        srand((unsigned) time(0));
        float p = prob[i];
        Matrix<int> A(1024, 1024);
        randomfill(A, p);
        A = A.pow(3);
        int sum = 0;
        for (int i = 0; i < A.nrows; i++){
            sum += A(i, i);
        }
        printf("number of triangle: %d\n", sum/6);
    }
    return 0;
}
```

Number of Triangles				
Probability	Theoretical Value	Trial I	Trial II	Trial III
0.01	178	174	150	180
0.02	1427	1726	1436	1273
0.03	4818	4531	4588	4900
0.04	11420	11466	11352	11399
0.05	22304	22608	22078	22795

The table demonstrates the result of number of triangles based on A^3, where A is the adjacency matrix of a randomly generated graph with $P(edge) = p$. We can see that our empirical results agree with the theoretical expectations.

Triangle Comparison